Automated Software Engineering (2023) 30:9
https://doi.org/10.1007/s10515-022-00373-7

Freeze-and-mutate: abnormal sample identification for DL applications through model core analysis

Huiyan Wang[1,2] · Ziqi Chen[1,2] · Chang Xu[1,2]

Received: 28 August 2022 / Accepted: 27 November 2022 / Published online: 18 January 2023

Abstract

Deep learning (DL) applications, representing an emerging form of new software, are gaining increasing popularity by their intelligent and adaptive services. However, their service reliability depends highly on the prediction accuracy of their internally-integrated DL models. In practice, DL models are often observed to suffer from ill predictions upon abnormal inputs (e.g., adversarial attacking samples, out-of-distribution (OOD) samples, and etc.), and this could easily lead to unexpected behaviors or even catastrophic consequences (e.g., system crash). One promising way to guard the application reliability is to reveal such abnormal inputs in time before they are fed to the DL models integrated in the concerned applications. Then remedy actions (e.g., discarding or fixing these inputs) can be done to protect applications from acting abnormally. Existing work addressed this revealing problem by either making sample distance-comparison based analysis or generating sufficient model mutants for comparative analysis. However, such treatments caused a restricted focus on samples only, while overlooking the DL models themselves, or had to analyze massive mutants, incurring non-negligible overheads to applications. In this article, we propose a novel approach, NetChopper, to conducting a core analysis on the target DL model, and then partitioning it into two parts, one associating closely with the training knowledge being the model core (expected to be important and thus stable), and the other being the remaining part (expected to be immaterial and thus changeable). Based on such partitioning, NetChopper proceeds to preserve (or freeze) the model core, but mutate the remaining part to produce only a small number of model mutants. Later, NetChopper becomes able to reveal abnormal inputs from normal ones by exploiting these model-relevant and light-weight mutants only. We experimentally evaluated NetChopper by widely-used DL subjects (e.g., MNIST+LeNet4, and CIFAR10+VGG16) and typical abnormal inputs (e.g., adversarial and OOD samples). The results reported NetChopper 's promising AUROC scores in revealing the abnormal degrees of inputs, generally and stably outperforming, or comparably effective as, state-of-the-art techniques (e.g., mMutant, Surprise, and Mahalanobis),

Extended author information available on the last page of the article

and also confirmed its high effectiveness and efficiency (with only marginal online overhead).

Keywords Deep learning models · Abnormal sample identification · Model core analyses · Freeze and mutate

1 Introduction

Modern applications have gained increasing popularities by deploying deep learning modules (typically by DL models) to support decision-making intelligence and adaptive services. This trend also represents an emerging form of new software (sometimes known as "into a Software 2.0 era Karpathy 2017"). Such applications are especially welcomed in many dynamically-interactive or human-intensive fields, e.g., image classification (He et al. 2016), object identification (Szegedy et al. 2013), natural language processing (Mikolov et al. 2010), autonomous driving (Bojarski et al. 2016), and etc. Those DL-based applications (or DL applications for short) have been typically developed with an integrated DL model for internal decision-making in a seamless prediction cooperation way. However, unlike Software 1.0 (Karpathy 2017) (traditional programs) that are almost fully written by human programmers and have clear logic flows specified and implemented, the working logic of Software 2.0 is implicitly embedded in numerous weight values of trained models, which are rather abstract and human-unfriendly for understanding and debugging. At present, the reliability of DL models depends highly on their prediction accuracy. Due to statistical characteristics of DL training, DL models are capable of predicting correctly for input samples generally in most scenarios, but they could still occasionally suffer from ill predictions for abnormal samples (Goodfellow et al. 2015), thus affecting applications' reliable deployment at runtime and thus leading to unexpected application misbehaviors or even catastrophic consequences (Dia 2021). This problem not only undermines DL models' general application in many, including safety-critical, scenarios such as autonomous driving and malware detection, but also severely incurs uncontrollable reliability problems to popular DL-based applications.

Existing work has studied this problem from the perspective of detecting adversarial samples, which are considered as one of typical abnormal sample sources. Adversarial samples could be generated by adding carefully crafted human-imperceptible perturbations into original clean samples deliberately, aiming to fool certain DL models' predictions (Goodfellow et al. 2015). Many adversarial attackers (e.g., FGSM Goodfellow et al. 2015, C &W Carlini and Wagner 2017, DF Moosavi-Dezfooli et al. 2016, and etc.) have exhibited a high success rate in fooling DL models and making them predict mistakenly. Besides this direction, another typical abnormal sample source could be out-of-distribution (OOD) samples. OOD samples, although not necessarily carrying attacking purposes, follow a distribution essentially different from the training samples that are associated with the target DL model, and can also make the model predict mistakenly. Such abnormal samples

(including both adversarial and OOD ones) thus call for effective techniques to recognize and isolate them from normal ones so as to defend the reliability of DL applications.

To address the abnormal sample detection problem, some existing work chose to measure and compare sample distances to distinguish them (Carrara et al. 2017; Sitawarin and Wagner 2019; Cohen et al. 2020). However, they could suffer from tedious and time-consuming distance comparisons and thus lack necessary efficiency (Bulusu et al. 2020). Some other work proposed to dig into input samples' incurred behaviors during their associated predictions, in order to recognize their subtle differences. For example, mMutant (Wang et al. 2019), one piece of state-of-the-art work along this research line, combines mutation analysis and DL testing to reveal abnormal samples' different label changing behaviors upon prepared mutants. However, it requires typically a large number of mutants and their analysis is very time-consuming, thus limiting its practical usages. Regarding such efficiency requirements, there is also one line of work (e.g., Surprise Kim et al. 2019 and Mahalanobis Lee et al. 2018) that examines an input sample's behavior through the original model's prediction from various granularities and inspects evidence for being anomaly, very light-weight and thus efficient. However, to collect such evidence, the existing work has to obtain a certain amount of abnormal samples in advance for preparation, e.g., for training an additional abnormal identification classifier, and this could sometimes be infeasible in practice.

In this article, we propose a novel approach, named NetChopper, to first conducting a core analysis on the given DL model, and then partitioning it into two parts. One part is supposed to associate closely with the model's training knowledge, named *model core* (expected to be important and thus *stable*), and the other is the remaining part (expected to be immaterial and thus *changeable*). Based on them, NetChopper chooses to freeze the model core and mutate the remaining part, thus producing a few model mutants. This process could be conducted in a model-related and lightweight way, resulting in a set of mutants useful for later analysis to distinguish the behaviors of abnormal samples from those of normal samples by prediction differential analysis. Thus, NetChopper 's core analysis and core-based mutation work in a generic way for model interpretation, potentially useful for model analysis and testing problems.

To evaluate NetChopper 's performance, we conducted experiments on two widely-used DL subjects (MNIST+LeNet4 and CIFAR10+VGG16), and observed that: (1) NetChopper 's reported abnormal scores on abnormal samples and normal samples indeed differed significantly, with AUROC scores consistently outperforming, or comparably effective as, existing techniques; (2) NetChopper achieved satisfactory and stable effectiveness in identifying abnormal samples, with an average F_1 score of 0.85 and average accuracy of 0.85, (3) NetChopper also exhibited promising efficiency by taking only 0.11-1.62 minutes during its offline preparation (small and acceptable) and 0.01–0.97 milliseconds (per sample) during its online sample identification (extremely efficient), much preferred than existing techniques.

The remainder of this article is organized as follows. Section 2 introduces necessary DL background. Section 3 presents our NetChopper approach for effectively identifying abnormal inputs for DL applications. Section 4 experimentally

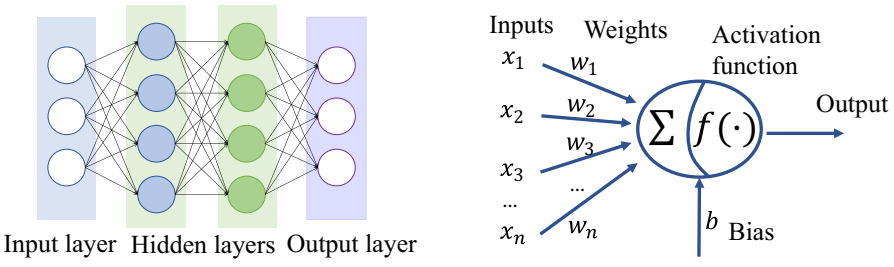
evaluates NetChopper 's performance, and compares it to state-of-the-art techniques on the effectiveness and efficiency. Then, Section 5 discusses NetChopper 's potential application scenarios from a software engineering perspective. Finally, Section 6 analyzes and compares related work in recent years, and Section 7 concludes this article.

2 Background

2.1 Deep learning

Deep learning is a sub-field of machine learning, which refers to using deep artificial neural networks (ANN) to learn representations of data. The most commonly used ANN is Deep neural network (DNN). A typical DNN consists of an input layer, an output layer, and multiple hidden layers, as shown in Fig. 1a. Each layer consists of multiple artificial neurons, following the M-P neuron model, as shown in Fig. 1b. Typically, deep learning includes training and prediction phases. The former one is basically responsible for assigning the most suitable learning parameters (i.e., weights and biases), typically using the backpropagation algorithm (Rumelhart et al. 1986). Then, in the prediction phase, when fed by any input sample from the input layer, each neuron would receive signals (i.e. output values) from the neurons in its previous layer, and carry out weighted summation and activation operations for the final prediction, based on assigned parameters in training.

We emphasize on the widely used convolutional neural networks (CNN), which have been popular in many fields, e.g., computer vision, natural language processing, and so on. In addition to the traditionally fully connected layers, a typical CNN usually has convolutional layers and pooling layers, which are mainly responsible for feature extraction and dimension reduction. The local receptive field and shared weights of the convolutional layers achieve the purpose of extracting features and reducing the number of parameters, and the pooling layer further reduces the feature dimension.



(a) Basic architecture of a DNN (b) M-P neuron model

Fig. 1 A DNN architecture and M-P neuron model

2.2 DL testing and abnormal sample identification

Although DL models perform excellently on many tasks, they are also observed to suffer from uncertainty (Gal and Ghahramani 2016) and non-testability (Murphy et al. 2007) problems. This is because different from traditional software, whose decision logic is clearly specified by developers, the decision logic of DL models is derived from a large number of trained parameters with low interpretability. The poor performance of a DL model in a certain application scenario can be attributed to a combination of factors, e.g., the design of model structure, training data, training parameters, or the application scenario is indeed abnormal to the model.

Therefore, analogous to the definition of machine learning (ML) bugs and ML testing in (Zhang et al. 2020), a *DL bug* can be defined as any imperfection in a DL item that causes a discordance between the existing and the required conditions. *DL testing* can be defined as the activity designed to reveal DL bugs. In DL testing, abnormal sample detection is a popular topic to help reveal DL bugs, since abnormal samples usually work beyond the handling ability of the model. In practice, *adversarial samples* and *out-of-distribution (OOD) samples* are two typical abnormal sample sources. An adversarial example is generated by adding carefully crafted human-imperceptible perturbations into original clean samples deliberately, aiming to fool DL models' predictions. An OOD sample is an input sample following a distribution different from the training samples that are associated with the target DL model, and can also make the model predict mistakenly. Many existing techniques are proposed to effectively identify such abnormal samples from different aspects (Lee et al. 2018; Xu et al. 2018; Metzen et al. 2017), though there are still limitations on the identification performance.

3 Our approach

3.1 Approach overview

Our approach consists of two phases, namely, an offline preparation phase and an online identification phase, as shown in Fig. 2. Let a trained model be M and its associated training samples be X.

NetChopper 's first phase would: (1) explore model M with samples X to prioritize M's different components (by focusing on the importance of neurons in M), (2) based on the prioritization, partition model M into two parts, one associated closely with the training knowledge (with respect to samples X), named *model core* (expected important and stable), and the other that remains from M (expected immaterial and changeable), we then freeze the model core and mutate the remaining part to produce a model mutant M', and finally (3) calculate the upper bound of training samples X's corresponding prediction differences between M and M' as an appropriate threshold to separate abnormal samples from normal samples.

With such preparations, for any new input sample, NetChopper can then conduct its abnormal sample identification. That is, for a given sample s, NetChopper 's second phase would: (1) profile sample s's prediction differences upon M and the

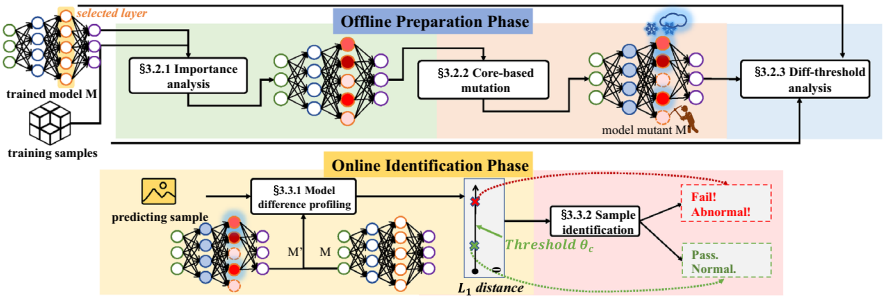


Fig. 2 NetChopper overview

mutant M', (2) with the pre-calculated cutting threshold θ_c, determine sample s to be "abnormal" ($> \theta_c$) or "normal" ($\leq \theta_c$).

We elaborate on the two phases in turn below.

3.2 Phase 1: Offline preparation phase

3.2.1 Neuron importance analysis

Considering a trained model M, and its associated training samples X, NetChopper would first conduct a neuron importance analysis upon a selected layer. NetChopper uses Layer-wise Relevance Propagation (LRP) (Montavon et al. 2019) method to calculate the importance of neurons. It propagates the prediction result backward in the neural network, by means of purposely designed local propagation rules. There are several different LRP propagation rules, e.g., LRP-0, LRP-ϵ, LRP-γ. NetChopper chooses LRP-ϵ to avoid noises and get sparser analysis results. More elaborately, for a sample x, let j and k be neurons at two consecutive layers of the neural network, neuron j's LRP relevance score R_j^x is given by:

$$R_j^x = \sum_k \frac{a_j^x w_{jk}}{\epsilon + \sum_{0,j} a_j^x w_{jk}} R_k^x, \tag{1}$$

where a_j^x represents the output of neuron j, w_{jk} represents the weight between neuron j and k, ϵ is set to 0.25 as recommended in (Montavon et al. 2019). For training samples X, NetChopper calculates the sum of absolute values of LRP relevance scores as the neuron importance score $Imp(j)$ of the dataset:

$$Imp(j) = \sum_{x \in X} |R_j^x| . \tag{2}$$

Except the input and output layer, NetChopper does not restrict which layer to select. We would study the effect of layer selections in our later evaluation.

For the model trained for classification tasks, NetChopper calculates neuron importance scores separately for the subset of each class to conduct a finer

analysis. In addition, when the training dataset is large, a randomly sampled subset of the training dataset can replace the entire dataset to improve efficiency. And for some practical cases without available training samples, NetChopper 's offline analysis could also be fed with generally normal samples, which are expected to be diverse and representative.

3.2.2 Core-based mutation

In the last step, we have measured neuron importance scores in M. Based on the prioritization of those neurons' importance scores, NetChopper would partition the model into two parts: one part associated closely with the training knowledge, named *model core* (expected to be important and thus stable), and the remaining part (expected to be immaterial and thus changeable). Generally, NetChopper would treat those neurons with high importance scores as "*core neurons*", and extract them into the model core. To do so, NetChopper would control the proportion of neurons in each layer that can be considered as core neurons in practice, by setting a built-in *core percentage* (p_{core}). Then, any neuron belonging to the model core would be "*frozen*", i.e., no further mutation allowed, in order to ensure that the core knowledge learned from training can be preserved during the mutation. In practice, p_{core} can be empirically set to gradually increase until a certain amount of training samples (<1%) no longer hold their original predictions on generated mutants. In our later evaluation, we would also investigate the impacts of different core percentages on NetChopper 's performance.

After freezing the model core, NetChopper then arbitrarily mutates neurons belonging to the remaining part, which are believed to be immaterial and thus changeable, and their diverse mutations are expected to help reveal different behaviors of abnormal samples.

One straightforward mutation treatment is to deactivate these neurons, i.e., set their outputs to be zero, suggesting removing their corresponding influences on the prediction. That is:

$$a'_j = \begin{cases} a_j & j \text{ is a core neuron,} \\ 0 & \text{otherwise. //mutate} \end{cases} \tag{3}$$

In practice, this model mutant can be implemented as inserting a mask layer after the selected layer of the original model, where the outputs of core neurons are multiplied by 1 and the rest by 0.

Note that, since we analyze neuron importance scores of each class for classification models as aforementioned, the number of model mutants should be equal to the number of classes theoretically. To reduce storage cost, when the number of classes is large, we can implement only one mutant and store the different weights of the mask layer, and replace the weights.

3.2.3 Diff-threshold analysis

For the original model M, let one generated model mutant be M'. In order to leverage the expected and unexpected behavioral differences between such two models to help identify abnormal samples, we investigate the prediction differences of obviously normal samples (i.e., training samples with correct predictions). Specifically, we measure the L_1 distance of model outputs between M and M' as the *abnormal score* of a input sample. For a sample x, we have:

$$Diff(x, M, M') = \left\| M(x) - M'(x) \right\|_1 . \tag{4}$$

We also observe that NetChopper 's importance calculation method are reasonable, i.e, the neurons we mutate do not have much effect on the normal samples' predictions in a preliminary experiment on a LeNet4 (LeCun et al. 1998) model with MNIST (Yann 1998) dataset (see 4.2). In this case, when we mutate 10% neurons in turn selected according to the increasing order of neuron's importance sores (from unimportant to important) and construct the corresponding mutants, the average *Diff*() score of 500 random training samples strictly increases. Especially, when we select to mutate the most important 10% of neurons, there is an obviously huge impact on the average *Diff*() score than other selections, suggesting only a small amount of neurons might dominate such predictions. This also suggests NetChopper 's reasonable ranking for the importance of neurons.

Note that, we do not directly adopt samples' label changing results like existing work (Wang et al. 2019), since NetChopper 's mutation is quite delicate and might not always propagate to the prediction label. After that, for all samples in M's training (i.e., X), we can obtain a series of their *Diff*() values. We then extract the *upper bound* of *Diff*() values as an appropriate threshold θ_c to distinguish between normal and abnormal samples. In order not to be affected by outliers, NetChopper refers to the calculation method of upper bound in drawing boxplots (Frigge et al. 1989). Therefore, instead of using abnormal samples to train an additional classifier like existing work (Kim et al. 2019; Lee et al. 2018), NetChopper obtains the thresholds automatically.

3.3 Phase 2: Online identification phase

After generating model mutant(s) (e.g., M'), and obtaining the corresponding threshold (e.g., θ_c), NetChopper can now conduct the online identification for any collected sample. Suppose an input sample x, and NetChopper would conduct the abnormal sample identification as follows.

3.3.1 Model difference profiling

For sample x, NetChopper can similarly collect its prediction difference i.e., $Diff(x, M, M')$ between the original model M and the model mutant M', as shown in Equation 4. For classification tasks, suppose the model predicts x as class i, the

difference is calculated between M and M'_i, i.e., the model mutant whose neuron importance is calculated by training samples from class i. It means that despite being predicted as class i, NetChopper can distinguish between normal class i samples and abnormal ones using the prediction differences. Therefore, unlike existing work (Wang et al. 2019) that feeds samples into a number of mutants, each sample only needs one model mutant to calculate the difference score in NetChopper.

3.3.2 Sample identification

After that, based on the obtained threshold for M' (i.e., θ_c), and the sample x's prediction difference $Diff(x, M, M')$, NetChopper can easily determine sample x to be "abnormal" ($Diff(x, M, M') > \theta_c$) or "normal" ($Diff(x, M, M') \leq \theta_c$).

4 Evaluation

In this section, we experimentally evaluate NetChopper and compare it to state-of-the-art techniques (i.e., mMutant, Surprise, and Mahalanobis) for their performance in identifying abnormal samples for DL applications.

4.1 Research questions

We aim to answer the following four research questions:

RQ1 (Distinguishing ability): How is NetChopper 's ability to distinguish between abnormal samples and normal samples effectively?

RQ2 (Detection effectiveness): How effective is NetChopper in identifying abnormal samples?

RQ3 (Controlling factors): How do NetChopper 's parameters (e.g., core percentage p_{core}, layer selection, scoring mechanism, and sampling selection) affect its effectiveness?

RQ4 (Efficiency): How efficient is NetChopper in identifying abnormal samples (during its offline preparation and online identification), as compared to existing techniques?

4.2 Experimental design and setup

We introduce experimental subjects, design, and implementation in turn below.

Experimental subjects We used two popular image classification datasets in the DL field as our experimental subjects, namely, MNIST (Yann 1998) and CIFAR10 (Krizhevsky and Hinton 2009), each associated with a state-of-the-art DL model, as shown in Table 1. *MNIST* is an image database for hand-written digit classification (with ten labels), which contains 60,000 training samples and 10,000 predicting samples for testing. Its associated DL model is *LeNet4* (LeCun et al. 1998). *CIFAR10* is another image database for object recognition (also with ten labels), which contains 50,000 training samples and 10,000 predicting samples. Its associated DL model

Table 1 Descriptions for datasets and associated DL models

Dataset	# samples	DL model	Accuracy
MNIST	60,000/10,000	LeNet4	98.52%
CIFAR10	50,000/10,000	VGG16	92.47%

is *VGG16* (Simonyan and Zisserman 2015). Within the scope of our experiments, we consider the predicting samples with correct predictions from each dataset as "normal" (as contrast to "abnormal" ones with incorrect predictions generated from adversarial attacking or OOD, as explained later soon).

In order to evaluate NetChopper 's performance on abnormal sample identification, for each dataset, we prepared two popular types of abnormal samples that have been intensively studied in existing work (Bulusu et al. 2020), i.e., adversarial samples (as *TYPE-I*), and out-of-distribution samples (as *TYPE-II*). TYPE-I abnormal samples could be constructed by adding carefully crafted human imperceptible perturbations into original samples deliberately, aiming to fool a DL model's prediction, while TYPE-II abnormal samples refer to those that are clearly under a different distribution from that of a DL model's training samples.

Regarding TYPE-I abnormal samples (adversarial), for each subject, we adopted five popular adversarial attackers upon its predicting samples with correct predictions, in order to deliberately fool the concerned model by changed predictions. These attackers are FGSM (Goodfellow et al. 2015), JSMA (Papernot et al. 2016), C &W (ℓ_2-norm) (Carlini and Wagner 2017), DF (Moosavi-Dezfooli et al. 2016), and BIM (Kurakin et al. 2017), used by following their typical settings. We either re-implemented existing adversarial attackers by following their publications or using their publicly available implementations if released. Concerning the detailed settings, since almost all such adversarial attackers inherited some internal controlling parameters, we also gave such related parameter settings in Table 2 (by following each attacker's general use suggestions). We

Table 2 Descriptions for TYPE-I abnormal samples

Subject	Attacker	Setups	Attacking success (%)
MNIST	FGSM	$\epsilon = 0.3$	98.63% (9,718/9,852)
	JSMA	$\theta = 1, \gamma = 0.1$	100% (9,852/9,852)
	C &W	binary_search_steps=10, max_iter=5000, initial_const=0.01	100% (9,852/9,852)
	DF	overshoot=0.02, max_iter=50	99.42% (9,795/9,852)
	BIM	$\epsilon = 0.3$, eps_step=0.03	100% (9,852/9,852)
CIFAR10	FGSM	$\epsilon = 0.02$	80.86% (7,477/9,247)
	JSMA	$\theta = 1, \gamma = 0.1$	100% (9,247/9,247)
	C &W	binary_search_steps=10, max_iter=5000, initial_const=0.01	100% (9,247/9,247)
	DF	overshoot=0.02, max_iter=50	99.94% (9,242/9,247)
	BIM	$\epsilon = 0.01$, eps_step=0.005	88.82% (82,13/9,247)

generate TYPE-I abnormal samples by selecting those samples that can originally be predicted correctly, but later be predicted incorrectly after the attacking. For example, considering the MNIST subject and the FGSM attacker, 9,718 abnormal samples were obtained based on all MNIST's predicting samples with correct predictions (9,852), suggesting an attacking success rate of 98.63%. To alleviate possible threats brought by our implementations, we have confirmed that our measured attacking success rates indeed met respective attacking abilities they previously claimed.

Regarding TYPE-II abnormal samples (OOD), for each subject, we adopted two OOD datasets that had been well studied in existing OOD research, namely, FASHION_MNIST (Xiao et al. 2017) (FMNIST for short) and EMNIST (Cohen et al. 2017) for MNIST, and CIFAR100 (Krizhevsky and Hinton 2009) and SVHN (Netzer et al. 2011) for CIFAR10. Note that we consider all predicting samples from these four OOD datasets as "abnormal", since they were from other datasets completely different from our subjects (designed even for totally different classification tasks). We give these details in Table 3.

Experimental design We design the following independent variables to control the experiments:

- *Subject* We used two subjects, each concerning a dataset and a DL model, namely, MNIST + LeNet4, and CIFAR10 + VGG16. When with no ambiguity, we refer to each by the dataset name only.
- *Abnormal type* For each subject, we generated both TYPE-I and TYPE-II abnormal samples as explained earlier. Concerning TYPE-I abnormal samples, we used five attackers, i.e., FGSM, JSMA, C &W, DF, and BIM. Concerning TYPE-II abnormal samples, we leveraged FMNIST and EMNIST for MNIST, and CIFAR100 and SVHN for CIFAR10, respectively.
- *Core percentage* (p_{core}) To investigate the impacts of NetChopper 's different core percentages in identifying model cores, we controlled p_{core} to take a value of 0.05, 0.10, ..., or 0.95, with a pace of 0.05.
- *Selected layer* We investigate the impacts of different layer selections for NetChopper. Note that for the convolution part of a CNN, we take a convolution block as a unit to mutate, i.e., we operate on the outputs of the selected block. For MNIST+LeNet4 which has two convolution blocks and two fully-connected layers, we tried all the block/layers except the input and layer, namely, block1, block2, and fc1. For CIFAR10+VGG16 which has five convolution blocks and

Table 3 Descriptions for TYPE-II abnormal samples

Subject	OOD	# samples	Description
MNIST	FMNIST	10,000	Clothing classification
	EMNIST	10,000	Digit/Letter classification
CIFAR10	CIFAR100	10,000	100 classes object recognition
	SVHN	10,000	Street view recognition

three fully-connected layers, we evenly selected three block/layers from different depths, namely, block2, block4, and fc1.

- *Techniques* We also compare NetChopper with three state-of-the-art abnormal sample identification techniques, namely, mMutant (Wang et al. 2019), Surprise (Kim et al. 2019), and Mahalanobis (Lee et al. 2018). Concerning *mMutant*, which originally has four variants, we adopted two of its variants (mMutant-NAI and mMutant-GF), as they exhibited the best performance (Wang et al. 2019). Concerning *Surprise*, which originally has two variants (Surprise-LSA and Surprise-DSA), we adopted Surprise-LSA since both variants exhibited comparable performance (Kim et al. 2019). *Mahalanobis* (Lee et al. 2018) was configured to use its original setting (Lee et al. 2018). We used original implementations of these three techniques, or slightly adapted them to identify samples into "abnormal" and "normal" two categories for experimental purposes.

The four techniques need some setups: (1) NetChopper needs to select layers for its model mutation. We selected block1 for MNIST and block4 for CIFAR10 to generate core-based model mutants. Besides, by default, we set p_{core} to be 0.95 for both subjects. We also investigated these factors' impacts in RQ3. (2) mMutant needs to configure a mutation degree. This parameter was set to 0.05 for MNIST and 0.005 for CIFAR10 as suggested (Wang et al. 2019). For each mMutant variant, 200 mutants were generated, and its parameters for MNIST and CIFAR10 on classifying abnormal and normal samples were as suggested (Wang et al. 2019). (3) Surprise (Kim et al. 2019) needs to set a variance threshold for removing neurons. It was set to 10^{-5} for MNIST as suggested (Wang et al. 2019). For CIFAR10, the suggested value 10^{-4} would cause "NaN" when calculating kernel density estimation score, so we had to expand this value to 10^{-3}. The original article does not clarify which kind of layer LSA is more suitable, so we similarly chose the same layer as NetChopper for its analysis. (4) Mahalanobis (Lee et al. 2018) needs to configure a noise magnitude for input enhancement. This parameter was set to 0 for simplicity. For layer selection, we chose all blocks' outputs to calculate Mahalanobis scores as suggested. (5) Both Surprise (Kim et al. 2019) and Mahalanobis (Lee et al. 2018) additionally need normal and abnormal samples in hand for training their internal classifiers, and thus we randomly selected 1,000 samples (around 10% as suggested Kim et al. 2019; Lee et al. 2018) for this purpose.

As mentioned earlier, we regard the samples with correct predictions and from original MNIST/CIFAR10's predicting samples as "normal", and those generated by TYPE-I/TYPE-II treatments as "abnormal". We mixed normal and abnormal samples together for experiments. For example, concerning the MNIST subject and the FGSM type, we mixed all normal samples (9,852) from MNIST's predicting samples with correct predictions and all generated TYPE-I abnormal samples with wrong predictions (9,718) by FGSM together.

Then, for evaluating these techniques' performance (effectiveness and efficiency), we design the following metrics.

For the effectiveness, we used the *area under the receiver operating characteristic* (AUROC) to measure the distinguishing ability of sample scores generated by convention. And we measured the detection effectiveness of NetChopper by

precision, *recall*, F_1-*score* and *prediction accuracy*, which are commonly used in machine learning field to measure the effectiveness of binary classifiers.

For the efficiency, we recorded the time spent by each technique on its offline preparation overhead and online identification overhead. For example, for NetChopper, its offline preparation overhead refers to the time spent on its model mutant preparation, and its online identification overhead refers to the time spent on its runtime sample analysis and identification based on the prepared model mutants.

Implementation We conducted all experiments on a Linux server with two Intel(R) Xeon(R) Gold 5118 CPU @2.30GHz, 10 GeForce RTX 2080Ti GPUs, and 384GB RAMs, running Ubuntu 16.04.

4.3 Experimental results and analyses

4.3.1 RQ1 (Distinguishing ability)

This question studies NetChopper 's ability to discriminate between abnormal samples and normal samples corresponding its reported scores reported (i.e., *Diff*() scores). For this, we used t-test to measure how significant such differences were in a statistical way. With the null hypothesis that "NetChopper generated *Diff*() scores with no significant difference between abnormal samples and normal samples", we obtained a series of *p*-values as shown in Table 4. From the table, one can observe that all data are far less than 0.05 for both types I and II cases. Thus one can safely reject this hypothesis at a 95% confidence level. That is, we believe NetChopper can produce significantly different scores between abnormal and normal samples.

Therefore, we answer RQ1 as follows: NetChopper *can produce significantly different scores between abnormal and normal samples.*

Table 4 t-test of NetChopper

MNIST			CIFAR10		
Description	*t*	*p*	Description	*t*	*p*
Normal	–	–	Normal	–	–
I (FGSM)	66.04	<0.01	I (FGSM)	52.61	<0.01
I (JSMA)	43.66	<0.01	I (JSMA)	140.76	<0.01
I (DF)	84.45	<0.01	I (DF)	105.68	<0.01
I (C &W)	73.66	<0.01	I (C &W)	128.01	<0.01
I (BIM)	15.02	<0.01	I (BIM)	2.75	<0.01
II (EMNIST)	39.72	<0.01	II (EMNIST)	76.3	<0.01
II (CIFAR100)	58.96	<0.01	II (SVHN)	71.91	<0.01

4.3.2 RQ2 (Effectiveness)

This question studies the ability to discriminate between abnormal samples and normal samples of the abnormal scores reported by NetChopper (i.e., *Diff*() scores), as compared to existing techniques. We calculated the abnormal scores on each mixed dataset using NetChopper and three existing techniques. We list the corresponding AUROC results in Table 5.

From the table, we observed and analyzed that: (1) in most cases, the NetChopper 's abnormal scores had a strong distinguishing ability towards abnormal samples, leading to an average AUROC of 0.9809 for MNIST and 0.8837 for CIFAR10. (2) compared to the other three techniques, on MNIST, NetChopper had a more consistent performance with AUROC all greater than 0.94, while the other methods performed extremely well on some datasets (e.g., mMutant-NAI got AUROC of 0.9993 on JSMA), but their worst values are as low as around 0.5. On CIFAR10, although overall Mahalanobis slightly outperformed us on most datasets, note that its abnormal scores were derived from the logistic regression classifier trained with abnormal samples, while NetChopper did not use any abnormal samples in advance. mMutant and Surprise also did pretty well in some data sets (e.g., JSMA and C &W), but considering their relatively large overhead as studied later, they may not be that reliable. Therefore, NetChopper had comparable performance with these comparison techniques on CIFAR10. (3) it is worth mentioning that all methods performed poorly on BIM samples with CIFAR10 model. This may be related to the structure of VGG16 and the attack intensity of BIM attack, which is worth further study.

Based on abnormal scores, NetChopper automatically calculates a threshold (θ_c) to determine whether an input sample is "abnormal" or "normal", as described in 3.2.3. This research question studies how effective NetChopper is in identifying abnormal samples. We measured it by calculating the precision, recall, F_1 score, and prediction accuracy of NetChopper on mixed datasets, as shown in Table 6.

From Table 6, we observed that NetChopper achieved a nice balance between the sample identification precision and recall, leading to a satisfactory F_1 score of 0.86–0.92 and accuracy of 0.85–0.91 for MNIST. For CIFAR10, NetChopper also performed well on 6 out of 7 datasets with a F_1 score of 0.73–0.92 and accuracy of 0.75–0.89. NetChopper only had a poor performance on BIM dataset, which was consistent with the result of RQ1 and has been explained. Generally, the detection results of NetChopper were consistent with the distinguishing ability in RQ1, which indicated that NetChopper 's threshold analysis method is effective.

Therefore, we answer RQ2 as follows: NetChopper *was generally effective in identifying abnormal samples from normal ones, with satisfied AUROC scores, as long as an average F_1 score of 0.85 and average accuracy of 0.85, outperforming the existing techniques.*

4.3.3 RQ3 (Controlling factors)

We next study how NetChopper 's effectiveness could be affected by its controlling factors (i.e., core percentage p_{core}, layer selection, scoring mechanism, and sampling selection for mutation).

Table 5 Distinguishing effectiveness (AUROC score) of NetChopper with comparisons (Color Table online)

Subject	Description	NetChopper	Surprise-LSA	mMutant-GF	mMutant-NAI	Mahalanobis
	I (FGSM)	0.9961	0.9840	0.8322	0.8659	0.6819
	I (JSMA)	0.9649	0.8508	0.9993	0.9986	0.8756
	I (DF)	0.9988	0.7380	0.9973	0.9967	0.6776
MNIST	I (C&W)	0.9966	0.7147	0.9985	0.9976	0.6926
	I (BIM)	0.9678	0.9597	0.4395	0.5502	0.9728
	II (EMNIST)	0.9446	0.8114	0.8931	0.8618	0.9147
	II (FMNIST)	0.9975	0.9953	0.8958	0.8746	0.9823
	I (FGSM)	0.8321	0.8243	0.7493	0.8262	0.8460
	I (JSMA)	0.9827	0.9659	0.9968	0.9939	0.9870
	I (DF)	0.9928	0.9997	0.9600	0.9517	0.9987
CIFAR10	I (C&W)	0.9771	0.9187	0.9992	0.9913	0.9554
	I (BIM)	0.5865	0.6654	0.6433	0.5981	0.6120
	II (CIFAR100)	0.9026	0.8562	0.8408	0.9129	0.9251
	II (SVHN)	0.9124	0.8745	0.8145	0.9503	0.9517

[1] the closer the color of a grid is to red, the higher the AUROC score, and the closer it is to green, the lower the score.

Table 6 Detection effectiveness of NetChopper with comparisons (Color Table online)

Subject	Description	NetChopper				Surprise-LSA				mMutant-GF				mMutant-NAI				Mahalanobis			
		p	r	F_1	acc	p	r	F_1	acc	p	r	F_1	acc	p	r	F_1	acc	p	r	F_1	acc
MNIST	I (FGSM)	0.85	1.00	0.92	0.91	0.94	0.91	0.93	0.93	0.79	0.73	0.76	0.77	0.80	0.75	0.77	0.78	0.00	0.00	0.00	0.50
	I (JSMA)	0.84	0.98	0.91	0.90	0.81	0.71	0.76	0.77	0.84	1.00	0.92	0.91	0.84	1.00	0.91	0.91	0.83	0.79	0.81	0.82
	I (DF)	0.85	1.00	0.92	0.91	0.69	0.61	0.65	0.67	0.84	1.00	0.92	0.91	0.84	1.00	0.91	0.91	0.74	0.61	0.67	0.70
	I (C&W)	0.85	1.00	0.92	0.91	0.66	0.60	0.63	0.65	0.84	1.00	0.92	0.91	0.84	1.00	0.91	0.91	0.78	0.54	0.64	0.69
	I (BIM)	0.84	0.98	0.91	0.90	0.90	0.88	0.89	0.89	0.40	0.12	0.19	0.47	0.38	0.11	0.17	0.46	0.90	0.95	0.93	0.93
	II (EMNIST)	0.83	0.89	0.86	0.85	0.82	0.64	0.72	0.74	0.82	0.84	0.83	0.83	0.81	0.81	0.81	0.81	0.90	0.79	0.84	0.85
	II (FMNIST)	0.85	1.00	0.92	0.91	0.98	0.96	0.97	0.97	0.82	0.83	0.82	0.82	0.81	0.79	0.80	0.80	0.96	0.92	0.94	0.94
CIFAR10	I (FGSM)	0.72	0.71	0.72	0.75	0.74	0.63	0.68	0.74	0.81	0.39	0.53	0.69	0.74	0.67	0.71	0.75	0.80	0.51	0.63	0.73
	I (JSMA)	0.82	1.00	0.90	0.89	0.90	0.90	0.90	0.90	0.93	1.00	0.96	0.96	0.84	1.00	0.92	0.91	0.91	0.96	0.93	0.93
	I (DF)	0.81	0.97	0.88	0.87	0.82	0.79	0.80	0.81	0.92	0.87	0.89	0.90	0.84	0.94	0.89	0.88	0.86	0.90	0.88	0.87
	I (C&W)	0.82	1.00	0.90	0.89	0.85	0.82	0.83	0.84	0.93	1.00	0.96	0.96	0.84	1.00	0.92	0.91	0.88	0.96	0.92	0.91
	I (BIM)	0.49	0.24	0.32	0.52	0.59	0.41	0.48	0.59	0.68	0.18	0.29	0.57	0.53	0.24	0.33	0.54	0.54	0.36	0.43	0.56
	II (CIFAR100)	0.81	0.89	0.85	0.83	0.79	0.77	0.78	0.77	0.88	0.54	0.67	0.72	0.84	0.88	0.86	0.85	0.85	0.83	0.84	0.84
	II (SVHN)	0.82	0.96	0.89	0.87	0.80	0.82	0.81	0.80	0.88	0.52	0.66	0.72	0.85	0.98	0.91	0.90	0.86	0.88	0.87	0.87

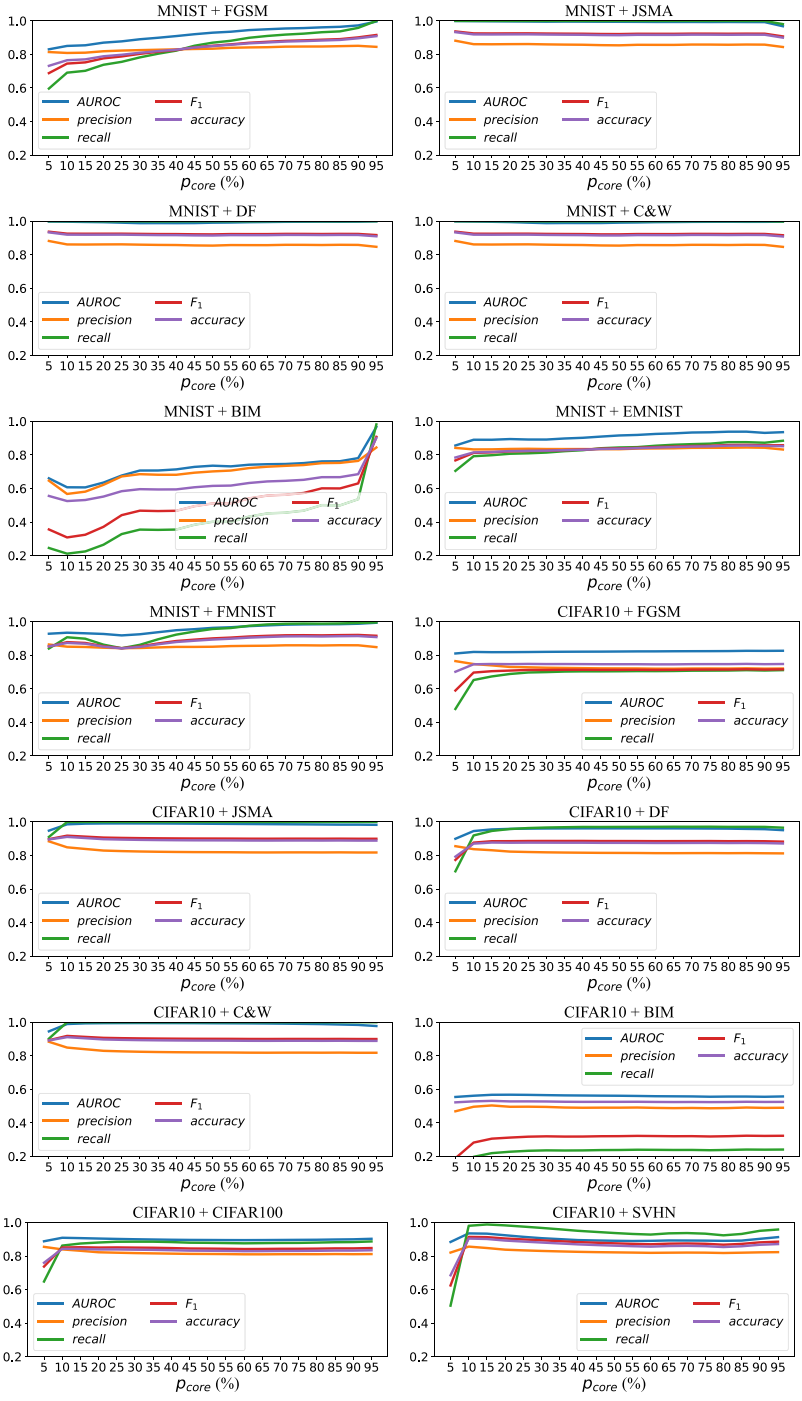


Fig. 3 Effectiveness results of different p_{core} values for MNIST and CIFAR10

Core percentage We controlled to set different values to NetChopper 's internal core percentage p_{core}, which denotes the proportion of core neurons preserved in the selected layer of the model, as well as affecting how many neurons could be left for the mutation to generate new model mutants. We list the results in Fig 3.

From the figure, we observed that in most cases, when p_{core} increased, suggesting that more model neurons belonging to "core" and would not be mutated, the corresponding effectiveness metrics generally increased steeply at first then became stable. This is understandable since when p_{core} was set to a small value, NetChopper 's mutation could happen to almost any neuron in the model, and this would bring uncontrollable impact to the model. Then when the p_{core} value increased, more core neurons were preserved, and correspondingly more core knowledge learned from training was preserved. This contributed to a controllable mutation and stable result. As such, the impact when the p_{core} value was small could be a little unpredictable. Still, when the p_{core} value was set over 60%, its trend started to behave similarly as expected. From this figure, we thus set 95% as the default p_{core} value for our both subjects.

By further investigating into the satisfactory p_{core} settings, we observe that in those cases, training samples are all able to hold their original predictions with an extremely high probability (no less than 99%), suggesting that the corresponding model core can still hold the knowledge for prediction effectively. Therefore, we do suggest that for a new scenario, in order to set a suitable p_{core} value, one can choose any value that is able to incur only marginal interruption upon training samples' predictions. We believe that this might be a suitable and automatic option for NetChopper 's practical application.

Layer selection Next, we consider different layers in DL models for selection in NetChopper. Tables 7 and 8 show the results of representative layers from different depths with respect to subject MNIST and CIFAR10. From these tables, we observe that: (1) on MNIST, the effectiveness of NetChopper was sensitive to the layer selections. For example, the F_1 score and accuracy on layer fc1 was significantly lower than those on block1 and block2. For FGSM and BIM, this sensitivity

Table 7 Effectiveness results of layer selections (MNIST)

Description	block1 $p/r/F_1/acc$	block2 $p/r/F_1/acc$	fc1 $p/r/F_1/acc$
I (FGSM)	0.84/1.00/**0.91/0.91**	0.84/0.94/0.89/0.88	0.64/0.41/0.50/0.59
I (JSMA)	0.84/0.97/0.90/0.89	0.85/1.00/**0.92/0.91**	0.71/0.56/0.63/0.67
I (DF)	0.84/1.00/0.91/**0.91**	0.85/1.00/**0.92/0.91**	0.70/0.53/0.60/0.65
I (C &W)	0.84/1.00/0.91/**0.91**	0.85/1.00/**0.92/0.91**	0.69/0.50/0.58/0.64
I (BIM)	0.84/0.99/**0.91/0.90**	0.73/0.49/0.59/0.66	0.52/0.25/0.34/0.51
II (EMNIST)	0.83/0.89/**0.86/0.85**	0.83/0.87/0.85/0.84	0.65/0.42/0.51/0.59
II (FMNIST)	0.84/1.00/**0.92/0.91**	0.85/0.99/**0.92/0.91**	0.64/0.39/0.49/0.58

[1] $p/r/F_1/a$ refers to *precision*/*recall*/F_1/*accuracy* values

The bold values denote the largestest values for F_1 and accuracy, respectively, across the three studied layer selections

Table 8 Effectiveness results of layer selections (CIFAR10)

Description	block2 $p/r/F_1/acc$	block4 $p/r/F_1/acc$	fc1 $p/r/F_1/acc$
I (FGSM)	0.77/0.70/**0.73/0.74**	0.77/0.70/**0.73/0.74**	0.78/0.69/**0.73/0.74**
I (JSMA)	0.82/1.00/0.90/0.89	0.82/1.00/0.90/0.89	0.83/1.00/**0.91/0.90**
I (DF)	0.82/1.00/0.90/0.89	0.82/1.00/0.90/0.89	0.83/1.00/**0.91/0.90**
I (C &W)	0.82/0.99/0.90/0.89	0.82/0.99/0.90/0.89	0.83/1.00/**0.91/0.90**
I (BIM)	0.57/0.28/**0.38/0.53**	0.57/0.28/**0.38/0.53**	0.57/0.26/0.36/**0.53**
II (CIFAR100)	0.82/0.88/**0.84/0.83**	0.82/0.88/**0.84/0.83**	0.82/0.85/**0.84/0.83**
II (SVHN)	0.83/0.95/0.88/**0.87**	0.83/0.95/0.88/**0.87**	0.84/0.94/**0.89/0.87**

The bold values denote the largestest values for F_1 and accuracy, respectively, across the three studied layer selections

was even more obvious. (2) on CIFAR10, however, the effectiveness of NetChopper seemed not sensitive to the layer selections, because the results were almost the same at different depths of layers in Table 8. This may be related to the more robust architecture of VGG16. (3) generally, for simple neural network architectures, a lower layer is preferred. For complicated neural network architectures, NetChopper does not restrict the layer selections.

Scoring mechanism To better investigate how NetChopper 's core analyses based on importance scoring truly contribute to its effectiveness, we additionally

Table 9 Detection effectiveness of NetChopper with/without scoring (pcore=0.8)

Subject	Description	NetChopper	NetC$^{\alpha}$	NetC$^{\beta}$
MNIST	I (FGSM)	0.9615	0.3052	0.8064±0.0218
	I (JSMA)	0.9933	0.1372	0.9699±0.0087
	I (DF)	0.9966	0.1890	0.9497±0.0119
	I (C &W)	0.9962	0.1822	0.9504±0.0121
	I (BIM)	0.7617	0.2481	0.4395±0.0261
	II (EMNIST)	0.9844	0.2680	0.8539±0.0156
	II (FMNIST)	0.9844	0.2344	0.8655±0.0380
CIFAR10	I (FGSM)	0.8247	0.6502	0.8241±0.0031
	I (JSMA)	0.9847	0.4237	0.9550±0.0034
	I (DF)	0.9600	0.5445	0.9055±0.0053
	I (C &W)	0.9887	0.4446	0.9186±0.0079
	I (BIM)	0.5568	0.5109	0.5602±0.0023
	II (CIFAR100)	0.8969	0.5939	0.9048±0.0036
	II (SVHN)	0.8904	0.2718	0.9151±0.0059

NetC$^{\alpha}$ refers to NetChopper with opposite scoring (i.e., treat the part of neurons with smallest *Imp* scores as core), and NetC$^{\beta}$ refers to that with random scoring (i.e., treat a random part of neurons as core)

modified its kernel scoring treatment with two typical ones, thus obtaining two variants NetC$^{\alpha}$ and NetC$^{\beta}$. NetC$^{\alpha}$ chooses the neurons with the smallest *Imp* scores as core, thus working exactly opposite to our NetChopper, while NetC$^{\beta}$ randomly chooses neurons as core. We controlled the three approaches to mutating the same amount of neurons and compared their eventual detection effectiveness correspondingly, as shown in Table 9.

From the table, we can observe that, our original NetChopper 's scoring contributes greatly to its detection effectiveness. If we reverse NetChopper 's scoring in identifying core neurons, i.e., choose neurons with the smallest *Imp* scores as core, our obtained NetC$^{\alpha}$ produced much worse effectiveness, with AUROC scores being only 0.1372–0.3052 for MNIST, and 0.2718–0.6502 for CIFAR10, while NetChopper originally achieved 0.7617–0.9966 for MNIST, and 0.5568–0.9887 for CIFAR10. When one chooses to randomly identify core neurons (represented by NetC$^{\beta}$), the corresponding effectiveness can be improved as compared to the worst NetC$^{\alpha}$, with AUROC being 0.4134–0.9786 for MNIST, and 0.5579–0.9584 for CIFAR10, still worse than our original NetChopper. Moreover, note that, NetC$^{\beta}$ may sometimes degrade into NetC$^{\alpha}$, when one accidentally chooses neurons with quite small importance scores as core, exhibiting NetC$^{\beta}$'s unpromising effectiveness somehow.

Practical sampling To facilitate NetChopper 's practical usages on large scenarios, we additionally investigate into its effectiveness when only part of training samples were used in NetChopper to identify the core neurons. To do so, we chose to use controlled proportions of training samples with respect to MNIST, CIFAR10, and another larger scenario TinyImageNet (accompanied by ResNet). Note that our sampling is balanced across different label classifications to avoid possible bias. Results are shown in Fig. 4.

From the figure, we observe that, for simple scenarios like MNIST and CIFAR10, taking less than 5% training samples into NetChopper 's analyses can already produce stable and comparable detection effectiveness, while for some relatively larger scenarios like TinyImageNet, such ratio may increase to around 15–20%. Still, we observe that in practical usage, training sampling can be optional, and thus we suggest that when considering resource budgets, taking

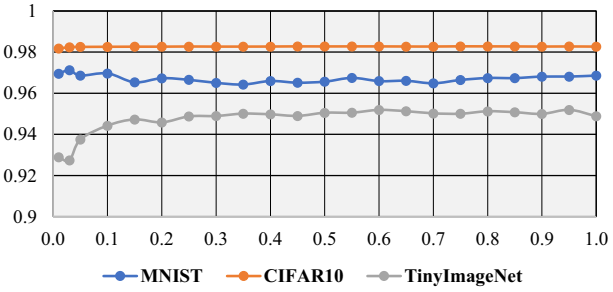


Fig. 4 Effectiveness comparisons of NetChopper with different sampling proportions

all training samples may be somehow infeasible for complex and large-dataset scenarios.

As a summary, we answer RQ3 as follows: NetChopper 's *internal factors could affect its effectiveness, and although they sometimes affect* NetChopper *'s stableness, its effectiveness generally holds.*

4.3.4 RQ4 (Efficiency)

We finally evaluate NetChopper 's efficiency in identifying abnormal samples by focusing on its time overheads in the offline mutation preparation and online identification, and compare them to those of the other three techniques. We give the results in Figs. 5 , 6.

From the figures, we observe that: (1) regarding the offline overhead, two mMutant variants took significantly more time (3.48 and 5.17 minutes for MNIST, and 23.80 and 7.70 minutes for CIFAR10) than the other three techniques (e.g., 0.11 and 1.62 minutes for NetChopper, 0.15 and 1.07 minutes for Surprise, and 0.05 and 0.13 minutes for Mahalanobis, on MINST and CIFAR10, respectively); (2) regarding

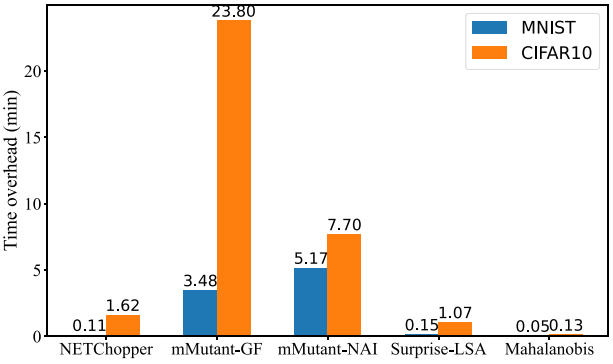


Fig. 5 Comparisons on offline overhead (min)

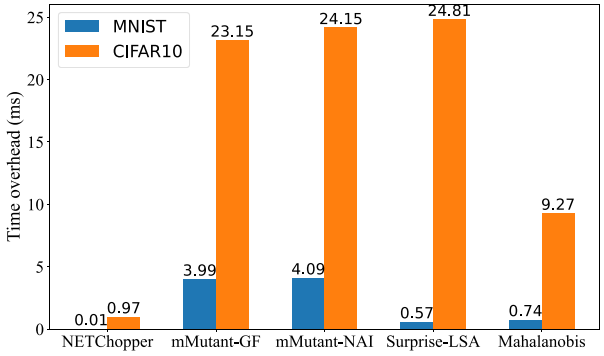


Fig. 6 Comparisons on online overhead (ms)

the online overhead (more importantly), NetChopper performed the best by costing significantly less time than the other three techniques, which only took 0.01 and 0.97 milliseconds for MNIST and CIFAR10 per sample. This is because for each sample, NetChopper only needs to feed it into two models (i.e., the original model and the model mutant and obtains their prediction difference to compare with the threshold θ_c. However, mMutant needs to feed it into hundreds of models. Surprise and Mahalanobis need to obtain and operate the intermediate outputs of the model. Therefore, they consumed more time than NetChopper. (3) summed up, NetChopper could be considered to both exhibit excellent efficiency (acceptable offline overhead and marginal online overhead);

Therefore, we answer RQ4 as follows: NetChopper *was very efficient with acceptable offline overhead and only marginal online overhead, exhibiting promising potential among studied techniques.*

4.4 Threat analysis and discussion

Our experiments used only two datasets, namely, MNIST and CIFAR10, and this might threaten the external validity of our experimental conclusions. We alleviated this threat as follows. First, our selected datasets have been typically used in relevant research, and for each dataset, we generated rich abnormal sample types, including five adversarial and two OOD ones, trying to make them diverse and representative. For comparisons, we selected three latest and highly related techniques (mMutant, Surprise, and Mahalanobis) for experimental effectiveness comparisons, representative as mutation analysis, distribution analysis, and distance analysis techniques (from both the SE and AI communities). We believe that our experimental conclusions should generally hold, and will test on diverse datasets and compare to more techniques in future.

5 Application discussion

We discussed some of potential application scenarios of our work, especially for SE, as follows.

Tolerating low-quality DL modules in software DL models are now being widely used as a vital module in modern software due to their intelligent and adaptive ability (sometimes known as "into a Software 2.0 era"), e.g., coping with dynamic-changing environments like recognize moving objects at runtime. However, DL models still do not promise to work perfectly especially for software's complex application scenarios at their deployments. Even if their performance could be acceptable for now, application scenarios keep evolving from time to time and that can easily cause these models to behave unexpectedly unsatisfactorily. Typically, when a piece of software is observed to be imperfect for deployment due to its inherited DL models, developers typically choose to refine exist-ing DL models by retraining it or replacing it with a new one. Either *Measuring DL model deployment's suitability for software* Considering modern

software with DL models, such software could be deployed in a complex scenario, which may not have been fully anticipated or tested in advance, since a full deployment and test can sometimes incur substantial re-source costs. Then, based on how many samples are identified as abnormal ones, NetChopper can be used for suggesting whether and how such software actually suits a specific scenario. More applications of our NetChopper technique include selecting and applying the most suitable DL model from a set of candidates during adapting a piece of software to a target scenario, as well as optimiz-ing the assignment of multiple DL models to multiple application scenarios.

Sample identification to promote DL-based software development Although we used only one application (picture labeling) in our evaluation, NetChopper can similarly apply to other fields since it works for DL classifiers, which are being widely used in many fields, i.e., natural language processing, malware detection, etc. Moreover, the identified samples (as abnormal ones) can also be used to guide optimization directions for refining DL models during the software development. For example, those identified abnormal inputs form a critical set for further consideration, which denotes those inputs that are indeed beyond a DL model's handling capability and would thus cause possi-bly misleading or incorrect predictions. Then developers can consider whether and how to use such identified inputs, e.g., for expanding the model's ability by retraining it with these particular input samples, or strengthening its original ability by keeping them isolated. More issues such as model stability and corner cases can also be taken into consideration during this refinement process, and such efforts can guide different optimization directions for such DL-based software.

Other contributions to the AI community Generally, NetChopper gives a quantity measurement for DL models' input samples, and this can also be helpful for the AI community. For example, by quantifying how inputs are expected or not for a DL model, this can give a suggestion on how to filter extremely out-of-scope samples, and this can also alleviate the cost of manual labelling for large-volume samples in new application scenarios. Moreover, DL models are traditionally compared by their prediction accuracies, considering input samples constructed for validation. With a NetChopper-alike wrapper for sample identification, the comparison can now have new considerations by combining their accuracies when accompanied by such a wrapper. For instance, suppose that models A and B have their original accuracies of 75% and 80% for a particular scenario. With NetChopper, their actual accuracies in practice could be instead improved to 90% and 85%, respectively. Then this may call for new research on how to understand a DL model's actual accuracy in practice and how to use such facts.

6 Related work

Our work mainly relates to work from three aspects, namely, DL testing, abnormal sample identification, and model comprehension and interpretation.

DL testing Testing has exhibited its unique importance on ensuring applications' reliability. Concerning DL testing, existing work mainly focuses on

proposing diverse test adequacy criteria and generating effective test inputs. Inspired by traditional code coverage criteria, Pei et al. (2017) first proposed neuron coverage (NC) in DL to measure the activated neurons by test inputs. Later, Ma et al. (2018) proposed DeepGuage, a more fine-grained testing criterion, to both measure a DL model's functionality and corner cases in neuron and layer level. Kim et al. (2019) then introduced surprise adequacy to calculate how surprising a test input is to a DL model with respect to its training dataset through kernel density estimation and distance-based methods. Gerasimou et al. (2020) proposed an importance-driven test adequacy criterion (IDC) to evaluate the semantic diversity of a test set. Along this line, some work proposes to effectively generate inputs to meet a higher coverage. Pei et al. (2017) used joint optimization to maximize NC while also exposing as many differences between multiple similar DL systems. Subsequent work (Guo et al. 2018; Tian et al. 2018; Xie et al. 2019) used coverage-guided fuzzing to generate inputs with various heuristic strategies. However, such inputs generated by these methods could be found unnatural (Zhang et al. 2018). However, there is also some debate on whether a higher coverage denotes better model reliability or robustness. For example, Li et al. (2019) pointed out that the structural coverage criteria could sometimes be misleading due to no strong correlation observed between misclassified inputs in a test set and their corresponding structural coverage metrics, and they (Li et al. 2020) also propose to debugging confidence errors for DNNs by operational calibration later. Other than testing DL models directly, another relevant line of work focuses on testing DL implementations by validating popular DL frameworks or APIs, since they are kernel to DL functionalities. For example, CRADLE (Pham et al. 2019) proposed a cross-validation framework for testing libraries from widely-used DL backends, e.g., tensorflow, pytorch, etc. Later, Wang et al. (2020) refined CRADLE for better performance by investigating and proposing ways to more effectively generate models for testing DL libraries via guided mutant generation. Other work (Wan et al. 2021, 2022) also directly or indirectly emphasized on the quality assurance for deep learning APIs used in DL programs.

Abnormal sample identification There is also a line of work, that emphasizes on identifying abnormal sample effectiveness, so as to ensure DL applications' ability at runtime. Early methods (Hendrycks and Gimpel 2016, 2017; Liang et al. 2018) focused on empirical differences between clean and perturbed samples, which are easy to bypass by new attacks (Carlini and Wagner 2017). Some other work chose to train abnormal sample detectors directly (Metzen et al. 2017; Gong et al. 2017; Feinman et al. 2017), therefore they usually required abnormal samples in advance for training. Some other work (Guo et al. 2018; Xu et al. 2018) tried to transform inputs to eliminate some impact of attacks, which could be model-independent and neglect the ability of a certain DL model. Lee et al. (2018) proposed to use confidence scores based on Mahalanobis distance from different layers, which was shown to be robust to typical abnormal samples, i.e., adversarial and OOD examples. However, its weighted averaging process still needs to train logistic regression detectors using both in-distribution and OOD samples. Wang et al. (2019) first proposed to use model mutation testing to detect adversarial samples, while it could suffer from huge time and space overhead.

This has also been further studied by Wang et al. (2020) by dissecting DNN networks efficiently. Our work also works along this line, trying to seek for a great balance on identification effectiveness and efficiency. These identification results could greatly contribute to the reliability maintenance for DL applications.

Model comprehension and interpretation Our work also relates to DNN models' comprehension and interpretation. Existing work typically analyzed internal units' specific contributions of DNN models from various perspectives. Some work used gradient-based (Zeiler and Fergus 2014; Mahendran and Vedaldi 2015) or up-convolutional techniques (Dosovitskiy and Brox 2016; Nguyen et al. 2017) to visualize hidden neurons as meaningful images to help users understand the role of individual neurons. Some other work (Ribeiro et al. 2016; Selvaraju et al. 2017; Fong and Vedaldi 2017) conducted sensitivity analysis on models, i.e., extract the important input regions or hidden neurons that are highly sensitive to the model outputs. Besides, there was another line of work (Pan and Rajan 2020) focusing on decomposing existing DNN models into sub-modules, so as to facilitate further application customization with more adapted functionalities. Some other work (LeCun et al. 1990; Hassibi et al. 1993; Molchanov et al. 2017) used sensitivity-based methods to prune unimportant parts of DNN models while maintaining the performance for computational acceleration. Our work also contributes to this line of research by adopting LRP (Montavon et al. 2019), an explanation technique for DNN which operates by propagating the prediction backward in the neural network according to local propagation rules, to calculate the importance scores of neurons. Based on the scores, we could further derive DNN models' core part that is associated closely to the training knowledge and model it as our *model core*. By doing so, NetChopper then presents a freeze-and-mutate mechanism by first combining model mutation and our model core analysis to effectively identify abnormal samples. We believe that NetChopper 's core analysis and core-based mutation could also suggest a generic way for model comprehension and interpretation.

7 Conclusion

DL applications' reliability is gaining popularity, and effectively identifying abnormal inputs for the DL models deployed in DL applications is a promising way towards the application reliability. In this article, we propose NetChopper to identify such abnormal inputs in an effective and efficient way, by generating a small number of valuable mutants via a novel core analysis and model mutation. The key insight is to isolate a model's core functionalities from marginal supports, and exploit such isolation to distinguish abnormal inputs from normal ones based on their behavioral differences, which also gives a generic way for model interpretation, potentially also useful for similar model analysis and testing problems. The experimental evaluation also confirmed NetChopper 's promising performance and stability, generally outperforming or comparably effective as existing techniques. In future, we consider to further strengthen NetChopper 's effectiveness by investigating more NetChopper 's variants and considering more complex application scenarios.

Author contributions HW and ZC wrote the main manuscript and conducted experiments in evaluation; CX reviewed the whole manuscript and response letter.

Funding This research was supported by the Natural Science Foundation of Jiangsu Province under Grant Nos. BK20202001 and BK20220771, and the Natural Science Foundation of China under Grant No. 61932021. The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

Declarations

Competing interests The authors declare no competing interests.

References

Apple. About face id advanced technology. [EB/OL]. https://support.apple.com/en-us/HT208108

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L.D., Monfort, M., Muller, U., Zhang, J., et al.: End to end learning for self-driving cars. *arXiv preprint*arXiv:1604.07316, (2016)

Bulusu, S., Kailkhura, B., Li, B., Varshney, P., Song, D.: Anomalous instance detection in deep learning: A survey. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), (2020)

Carlini, N., Wagner, D.: Adversarial examples are not easily detected: Bypassing ten detection methods. In: Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security, pages 3–14, (2017)

Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: Proceedings of the 38th IEEE Symposium on Security and Privacy (SP 2017), pages 39–57. IEEE, (2017)

Carrara, F., Falchi, F., Caldelli, R., Amato, G., Fumarola, R., Becarelli, R.: Detecting adversarial example attacks to deep neural networks. In: Proceedings of the 15th International Workshop on Content-Based Multimedia Indexing, pages 1–7, (2017)

Cohen, G., Afshar, S., Tapson, J., Van Schaik, A.: EMNIST: Extending mnist to handwritten letters. In: 2017 International Joint Conference on Neural Networks (IJCNN), pages 2921–2926. IEEE, (2017)

Cohen, G., Sapiro, G., Giryes, R.: Detecting adversarial samples using influence functions and nearest neighbors. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pages 14453–14462, (2020)

Dia, H.: 'self-driving' cars are still a long way off. here are three reasons why. [EB/OL]. https://theconversation.com/self-driving-cars-are-still-a-long-way-off-here-are-three-reasons-why-159234 Accessed April 22, (2021)

Dosovitskiy, A., Brox, T.: Inverting visual representations with convolutional networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pages 4829–4837, (2016)

Feinman, R., Curtin, R.R., Shintre, S., Gardner, A.B.: Detecting adversarial samples from artifacts. *arXiv preprint*arXiv:1703.00410, (2017)

Fong, R.C., Vedaldi, A.: Interpretable explanations of black boxes by meaningful perturbation. In: Proceedings of the IEEE international conference on computer vision, pages 3429–3437, (2017)

Frigge, M., Hoaglin, D.C., Iglewicz, B.: Some implementations of the boxplot. Am. Stat. **43**(1), 50–54 (1989)

Gal, Y., Ghahramani, Z.: Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In: International conference on machine learning, pages 1050–1059. PMLR, (2016)

Gerasimou, S., Eniser, H.F., Sen, A., Cakan, A.: Importance-driven deep learning system testing. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE 2020), pages 702–713. IEEE, (2020)

Gong, Z., Wang, W., Ku, W.-S.: Adversarial and clean data are not twins. *arXiv preprint*arXiv:1704.04960, (2017)

Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: Yoshua B., Yann L., (eds), 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, (2015)

Google. Google translate. [EB/OL]. https://translate.google.cn/

Guo, J., Jiang, Y., Zhao, Y., Chen, Q., Sun, J.: DLFuzz: differential fuzzing testing of deep learning systems. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), pages 739–743. ACM, (2018)

Guo, C., Rana, M., Cissé, M., van der Maaten, L.: Countering adversarial images using input transformations. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, (2018)

Hassibi, B., Stork, D.G., Wolff, G.J.: Optimal brain surgeon and general network pruning. In: IEEE international conference on neural networks, pages 293–299. IEEE, (1993)

He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016), pages 770–778, (2016)

Hendrycks, D., Gimpel, K.: A baseline for detecting misclassified and out-of-distribution examples in neural networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, (2017)

Hendrycks, D., Gimpel, K.: Visible progress on adversarial images and a new saliency map. (2016)

Karpathy, A.: Software 2.0. https://karpathy.medium.com/software-2-0-a64152b37c35, (2017)

Kim, J., Feldt, R., Yoo, S.: Guiding deep learning system testing using surprise adequacy. In: Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE 2019), (2019)

Krizhevsky, A., Hinton, G.: Learning multiple layers of features from tiny images Technical report. Citeseer, Princeton (2009)

Kurakin, A., Goodfellow, I.J., Bengio, S.: Adversarial machine learning at scale. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, (2017)

LeCun, Y., Denker, J.S., Solla, S.A.: Optimal brain damage. In: Advances in neural information processing systems, pages 598–605, (1990)

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)

Lee, K., Lee, K., Lee, H., Shin, J.: A simple unified framework for detecting out-of-distribution samples and adversarial attacks. In: Bengio, S., Wallach, H., Larochelle,H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds) Advances in Neural Information Processing Systems 31 (NIPS 2018), pages 7167–7177. Curran Associates, Inc., (2018)

Li, Z., Ma, X., Xu, C., Cao, C.: Structural coverage criteria for neural networks could be misleading. In: Proceedings of the 41th ACM/IEEE International Conference on Software Engineering (ICSE 2019 NIER), pages 269–280, (2019)

Li, Z., Ma, X., Xu, C., Xu, J., Cao, C., Lu, J.: Operational calibration: Debugging confidence errors for dnns in the field. In: Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), pages 901–913. ACM, (2020)

Liang, S., Li, Y., Srikant, R.: Enhancing the reliability of out-of-distribution image detection in neural networks. (2018)

Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y., et al.: DeepGauge: multi-granularity testing criteria for deep learning systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), pages 120–131. ACM, (2018)

Mahendran, A., Vedaldi, A.: Understanding deep image representations by inverting them. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pages 5188–5196, (2015)

Metzen, J.H., Genewein, T., Fischer, V., Bischoff, B.: On detecting adversarial perturbations. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, (2017)

Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., Khudanpur, S.: Recurrent neural network based language model. In: Eleventh annual conference of the international speech communication association, (2010)

Molchanov, P., Tyree, S., Karras, T., Aila, T., Kautz, J.: Pruning convolutional neural networks for resource efficient inference. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, (2017)

Montavon, G., Binder, A., Lapuschkin, S., Samek, W., Müller, K.R.: Layer-wise relevance propagation: an overview. Explainable AI: interpreting, explaining and visualizing deep learning, pages 193–209, (2019)

Moosavi-Dezfooli, S.M., Fawzi, A., Frossard, P.: DeepFool: a simple and accurate method to fool deep neural networks. In: Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2016), pages 2574–2582, (2016)

Murphy, C., Kaiser, G.E., Arias, M.: An approach to software testing of machine learning applications. In: Proceedings of the Nineteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2007), Boston, Massachusetts, USA, July 9-11, 2007, page 167. Knowledge Systems Institute Graduate School, (2007)

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., Ng, A.Y.: Reading digits in natural images with unsupervised feature learning. (2011)

Nguyen, A., Clune, J., Bengio, Y., Dosovitskiy, A., Yosinski, J.: Plug & play generative networks: conditional iterative generation of images in latent space. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4467–4477, (2017)

Pan, R., Rajan, H.: On decomposing a deep neural network into modules. In: Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), pages 889–900. ACM, (2020)

Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z., Swami, A.: The limitations of deep learning in adversarial settings. In: Proceedings of the 2016 IEEE European Symposium on Security and Privacy (EuroS &P 2016), pages 372–387. IEEE, (2016)

Papers with code. Adversarial defense. [EB/OL]. https://paperswithcode.com/task/adversarial-defense

Papers with code. Out-of-distribution detection. [EB/OL]. https://paperswithcode.com/task/out-of-distribution-detection

Pei, K., Cao, Y., Yang, J., Jana, S.: Deepxplore: Automated whitebox testing of deep learning systems. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017), pages 1–18. ACM, (2017)

Pham, H.V., Lutellier, T., Qi, W., Tan, L.: CRADLE: Cross-backend validation to detect and localize bugs in deep learning libraries. In: Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019), pages 1027–1038, (2019)

Ribeiro, M.T., Singh, S., Guestrin, C.: "why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining, pages 1135–1144, (2016)

Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back propagating errors. Nature **323**(6088), 533–536 (1986)

Selvaraju, R.R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., Batra, D.: Grad-cam: Visual explanations from deep networks via gradient-based localization. In: Proceedings of the IEEE international conference on computer vision, pages 618–626, (2017)

Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. In: Yoshua, B. and Yann, L. (eds) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, (2015)

Sitawarin, C., Wagner, D.: Defending against adversarial examples with k-nearest neighbor. *arXiv e-prints*, pages arXiv–1906, (2019)

Szegedy, C., Toshev, A., Erhan, D.: Deep neural networks for object detection. In: Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (eds) Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States, pages 2553–2561, (2013)

Tesla. Autopilot. [EB/OL]. https://www.tesla.com/autopilotAI

Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), pages 303–314. ACM, (2018)

Wan, C., Liu, S., Hoffmann, H., Maire, M., Lu, S.: Are machine learning cloud apis used correctly? In: Proceedings of the 43th International Conference on Software Engineering (ICSE 2021), pages 125–137. ACM, (2021)

Wan, C., Liu, S., Xie, S., Liu, Y., Hoffmann, H., Maire, M., Lu, S.: Automated testing of software that uses machine learning apis. In: Proceedings of the 44th International Conference on Software Engineering (ICSE 2022). ACM, (2022)

Wang, J., Dong, G., Sun, J., Wang, X., Zhang, P.: Adversarial sample detection for deep neural network through model mutation testing. In: Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE 2019), pages 1245–1256, (2019)

Wang, H., Xu, J., Xu, C., Ma, X., Lu, J.: DISSECTOR: input validation for deep learning applications by crossing-layer dissection. In: Proceedings of the 42th ACM/IEEE International Conference on Software Engineering (ICSE 2020), pages 727–738, (2020)

Wang, Z., Yan, M., Chen, J., Liu, S., Zhang, D.: Deep learning library testing via effective model generation. In: Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020), pages 788–799. ACM, (2020)

Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint*arXiv:1708.07747, (2017)

Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J., See, S.: Deephunter: a coverage-guided fuzz testing framework for deep neural networks. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 146–157, (2019)

Xu, W., Evans, D., Qi, Y.: Feature squeezing: detecting adversarial examples in deep neural networks. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society, (2018)

Yann L. The mnist database of handwritten digits. http://yann.lecun.com/exdb/mnist/, (1998)

Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: European conference on computer vision, pages 818–833. Springer, (2014)

Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering, pages 1–1, (2020)

Zhang, M., Zhang, Y., Zhang, L., Liu, C., Khurshid, S.: DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018), pages 132–142. ACM, (2018)

Authors and Affiliations

Huiyan Wang[1,2] · Ziqi Chen[1,2] · Chang Xu[1,2]

✉ Huiyan Wang
why@nju.edu.cn

Ziqi Chen
rubychen0611@gmail.com

Chang Xu
changxu@nju.edu.cn

[1] State Key Laboratory for Novel Software Technology, Nanjing University, 210023 Nanjing, Jiangsu, China

[2] Department of Computer Science and Technology, Nanjing University, 210023 Nanjing, Jiangsu, China