ArchRepair: Block-Level Architecture-Oriented Repairing for Deep Neural Networks

HUA QI[†], Kyushu University, Japan
ZHIJIE WANG[†], University of Alberta, Canada
QING GUO[*], Center for Frontier AI Research (CFAR), A*STAR, Singapore
JIANLANG CHEN, Kyushu University, Japan
FELIX JUEFEI-XU[‡], Meta AI, USA
FUYUAN ZHANG, Kyushu University, Japan
LEI MA[*], University of Alberta, Canada & The University of Tokyo, Japan
JIANJUN ZHAO, Kyushu University, Japan

Over the past few years, deep neural networks (DNNs) have achieved tremendous success and have been continuously applied in many application domains. However, during the practical deployment in industrial tasks, DNNs are found to be erroneous-prone due to various reasons such as overfitting, and lacking of robustness to real-world corruptions during practical usage. To address these challenges, many recent attempts have been made to repair DNNs for version updates under practical operational contexts by updating weights (*i.e.*, network parameters) through retraining, fine-tuning, or direct weight fixing at a neural level. Nevertheless, existing solutions often neglect the effects of neural network architecture and weight relationships across neurons and layers. In this work, as the first attempt, we initiate to repair DNNs by jointly optimizing the architecture and weights at a higher (*i.e.*, block) level.

We first perform empirical studies to investigate the limitation of whole network-level and layer-level repairing, which motivates us to explore a novel repairing direction for DNN repair at the block level. To this end, we need to further consider techniques to address two key technical challenges, *i.e.*, *block localization*, where we should localize the targeted block that we need to fix; and how to perform *joint architecture and weight repairing*. Specifically, we first propose *adversarial-aware spectrum analysis for vulnerable block localization* that considers the neurons' status and weights' gradients in blocks during the forward and backward processes, which enables more accurate candidate block localization for repairing even under a few examples. Then, we further propose the *architecture-oriented search-based repairing* that relaxes the targeted block to a continuous repairing search space at higher deep feature levels. By jointly optimizing the architecture and weights in that space, we can identify a much better block architecture. We implement our proposed repairing techniques as a tool, named *ArchRepair*, and conduct extensive experiments to validate the proposed method. The results show that our method can not only repair but also enhance accuracy & robustness, outperforming the state-of-the-art DNN repair techniques.

[†] Both authors contributed equally to this research.

[*] Corresponding authors: Qing Guo (tsingqguo@ieee.org) and Lei Ma (ma.lei@acm.org).

[‡] The work was done prior to joining Meta AI.

Authors' addresses: Hua Qi[†], Kyushu University, Fukuoka, Japan; Zhijie Wang[†], University of Alberta, Edmonton, Canada; Qing Guo[*], Center for Frontier AI Research (CFAR), A*STAR, Singapore; Jianlang Chen, Kyushu University, Fukuoka, Japan; Felix Juefei-Xu[‡], Meta AI, New York, USA; Fuyuan Zhang, Kyushu University, Fukuoka, Japan; Lei Ma[*], University of Alberta, Canada & and The University of Tokyo, Japan; Jianjun Zhao, Kyushu University, Fukuoka, Japan.

XXXX-XXXX/2023/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

CCS Concepts: • **Software and its engineering → Search-based software engineering**; • **Computing methodologies** → *Neural networks*; Computer vision; Supervised learning by classification.

Additional Key Words and Phrases: Deep Learning, DNN Repair, Neural Architecture Search

ACM Reference Format:
Hua Qi†, Zhijie Wang†, Qing Guo*, Jianlang Chen, Felix Juefei-Xu‡, Fuyuan Zhang, Lei Ma*, and Jianjun Zhao. 2023. *ArchRepair*: Block-Level Architecture-Oriented Repairing for Deep Neural Networks. 1, 1 (February 2023), 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern high-capacity deep neural networks (DNNs) have achieved astounding performance in many automated computer vision tasks ranging from complex scene understanding for autonomous driving [5, 8, 35, 38, 48, 62], to accurate DeepFake media detection [11, 29]; from challenging medical imagery grading and diagnosis [7, 14, 58, 67], to billion-scale consumer applications such as the face authentication for mobile payment, *etc*. Many of the tasks are safety- and mission-critical and the reliability of the deployed DNNs is of utmost importance. However, over the years, we have come to realize that the existence of unintentional (natural degradation corruptions) and intentional (adversarial perturbations) examples such as [6, 7, 15–17, 20–22, 27, 36, 58, 59, 63, 69] is a stark reminder that DNNs are vulnerable.

To tackle the DNN's vulnerability issues, many researchers have resorted to DNN repairing that aims at fixing the faulty DNN weights with the guidance of some specific repairing optimization criteria. An analogy to this is the traditional software repairing in the software engineering literature [19]. However, general-purpose DNN repairing may not always be feasible in practice, due to (1) the difficulty of generalizing DNNs to any arbitrary unseen scenarios, and (2) the difficulty of generalizing DNNs to seen scenarios but with unpredictable, volatile, and ever-changing deployed environment. For these reasons, a more practical DNN repairing strategy is to work under some assumptions of practical contexts and to perform task-specific and environment-aware DNN repairing where the model gap is closed up for a certain scenario/environment, or a set of scenarios/environments.

Compared to existing DNN repair work (*e.g.*, [18, 43, 50, 55, 68, 70]), this work takes the DNN repairing to a whole new level, quite literally, where we are performing **block-level** architecture-oriented repairing as opposed to network-level, layer-level, and neuron-level repairing. As we will show in the following sections that block-level repairing, being a midpoint sweet spot in terms of network module granularity, offers a good trade-off between network accuracy and time consumption for that just repairing some specific weights in a layer neglects the relationship between different layers while repairing the whole network weights leads to high cost. In addition, block-level repairing allows us to locally adjust not only the weights but also the network architecture within the block very effectively and efficiently.

To this end, as the first attempt, we repair DNNs by jointly optimizing the architecture and weights at the block level in this work. The modern block structure stems from the philosophy of VGG nets [53] and is generalized to a common designing strategy in the state-of-the-art architectures [24] (*e.g.*, ResNet) and optimization method [37]. To validate its importance for block-level repairing, we first study the drawbacks of network-level and layer-level repairing, which motivates us to explore a novel research granularity and repairing direction. Eventually, we identified that block-level architecture-oriented DNN repair is a promising direction. In order to achieve this, we need to address two challenges, *i.e.*, *block localization* and *joint architecture and weight repairing*. For the first challenge, we propose the *adversarial-aware spectrum analysis for vulnerable block localization* that considers the neuron suspiciousness and weights' gradients in blocks during the forward

and backward processes when evaluating a series of examples. This method enables more precise block localization even under few-shot examples. In terms of the second challenge, we propose the *architecture-oriented search-based repairing* that relaxes the targeted block to a continuous search space. The space consists of several nodes and edges where the node represents deep features and the edge is an operation to connect two nodes. By jointly optimizing the architecture and weights in that space, our method is able to find a much better block architecture for a specific repairing target. We conduct extensive experiments to validate the proposed repairing method and find that our method can not only enhance the accuracy but also the robustness across various corruptions. The different DNN models repaired with our technique perform better than the original one on both clean and corrupted data, with an average 3.939% improvement on clean data and 7.79% improvement on corrupted data, establishing vigorous general repairing capability on most of the DNN architectures.

Overall, the key contribution of this paper is summarized as follows:

- We propose block-level architecture-oriented repairing for DNN repair. The intuition of block structure design in modern DNNs provides a suitable granularity of DNN repair at the block-level [24]. In addition, we also show that jointly optimizing architecture and weights further brings the advantage of DNN repair over repairing DNN by only updating weights, which is demonstrated by our comparative evaluation in the experimental section.
- In terms of the *novelty and potential impacts*, existing DNN repair methods [13, 18, 43, 50, 55, 70] mostly focus on only repairing DNN via updating its weights while ignoring inherent DNN architecture design (*e.g.*, block structure and relationships between different layers), which could also impact the DNN behavior, whereas only repairing the weights could not address such an issue. Therefore, compared with existing work, this paper initiates a new and wide direction for DNN repair by taking relationships of DNN architecture design as well as layers and weights into consideration.
- Technically, we originally propose the adversarial-aware spectrum analysis-based block localization and architecture-oriented search-based repairing method, both of which are novel for DNN repair. The first one enables us to localize a vulnerable block accurately even with only a few examples. The latter formulates the repairing problem as the joint optimization of both the architecture and weights at the block level.
- We implement our repairing techniques in the tool *ArchRepair* and perform extensive evaluation against 6 state-of-the-art DNN repair techniques under 4 DNNs with different architectures on two different datasets. The results demonstrate the advantage of *ArchRepair* in achieving SOTA repairing performance in terms of both accuracy and robustness.

To the best of our knowledge, this is the very first attempt to consider the DNN repairing problem at the block level that repairs both network weights and architecture jointly. The results of this paper demonstrate the limitation of repairing DNN by only updating the weights, and show that other important DNN development elements such as architecture that encodes more advanced relationships of neurons and layers should also be taken into consideration during the design of DNN repair techniques.

2 DNN REPAIRING AND MOTIVATION

In this section, we review existing repairing methods in DNN and motivate our method. In Sec. 2.1, we thoroughly analyze previous DNN repair techniques from the viewpoint of different repairing targets, *e.g.*, the parameters (*i.e.*, weights) of the whole network, layers, or neurons. To this end, we formulate the core mechanism and compare the strengths and weaknesses of existing repairing

techniques, which inspires and motivates us to develop the block-level repairing method. To validate our motivation, we perform a preliminary study in Sec. 2.2.

2.1 DNN Repairing Techniques

In the standard training process, given a training dataset, we can train a DNN denoted as $\phi_{(\mathcal{W},\mathcal{A})}$ where $\mathcal{A}$ represents the network architecture related parameters determining what operations (*e.g.*, convolution layer, pooling layer, *etc.*) are used in the architecture, and $\mathcal{W}$ is the respective weights (*i.e.*, parameters of different operations). Generally, the architecture $\mathcal{A}$ is pre-defined and fixed during the training and testing processes. The variable $\mathcal{W}$ consists of weights for different layers.

Although existing DNNs (*e.g.*, ResNet [24]) have achieved significantly high accuracy on popular datasets, incorrect behaviors are always found in these models when we deploy them in the real world or test them on challenging datasets. There are a series of works that study how to repair these DNNs to be generalizable to misclassified examples, challenging corruptions, or bias errors [50, 55, 60, 68]. In general, we can formulate the existing repairing methods as

$$\mathcal{W}^* = \text{Locator}(\phi_{(\mathcal{W},\mathcal{A})}, \mathcal{D}^{\text{repair}}) \tag{1}$$

$$\hat{\mathcal{W}}^* = \arg\min_{\mathcal{W}^*} \text{J}(\phi_{(\mathcal{W}^*,\mathcal{A})}, \mathcal{D}^{\text{repair}}) \tag{2}$$

where $\mathcal{W}^*$ is a subset of $\mathcal{W}$ and $\hat{\mathcal{W}}^*$ is the fixed counterpart of $\mathcal{W}^*$. The dataset $\mathcal{D}^{\text{repair}}$ contains the examples for repairing guidance. Different works may set different $\mathcal{D}^{\text{repair}}$ according to the repairing scenarios. For example, Yu *et al.* [68] sets $\mathcal{D}^{\text{repair}}$ as the combination of the augmented training dataset. We will show that our method can address different repairing scenarios. Intuitively, Eq. (1) is to find the weights we need to fix in the DNN, and Eq. (2) with a task-related objective function $\text{J}(\cdot)$ is to fix the selected weights $\mathcal{W}^*$ and produce a new one $\hat{\mathcal{W}}^*$.

The above formulation can represent a series of existing repairing methods. For example, when we try to fix all weights of a DNN (*i.e.*, $\mathcal{W}^* = \mathcal{W}$) and set the objective function $\text{J}(\cdot)$ as the task-related loss function (*e.g.*, cross-entropy function for image classification) with different data augmentation techniques on collected failure cases as $\mathcal{D}^{\text{repair}}$ to retrain the weights, we actually get the methods proposed by [50] and [68]. In addition, when we employ the gradient loss of weights and forward impact to localize the targeted weights and use a fitness function to fix localized weights, the formulation becomes the method [55].

Nevertheless, with the general formulation in Eq. (1) and Eq. (2), we can see that existing repairing methods have the following limitations:

- Existing works only fix the targeted DNN either at the network-level (*i.e.*, fixing all weights of the DNN) or at the neuron-level (*i.e.*, only fixing partial weights of the DNN), and ignore the effects of the architecture $\mathcal{A}$.
- Only repairing some specific weights in a layer could easily neglect the relationship between different layers while repairing the whole network's weights leads to high costs.

Note that, the state-of-the-art DNNs (*e.g.*, ResNet [24]) are often made up of several blocks where each block is built with stacked convolutional and activation layers. Such block-like architecture is mainly inspired by the philosophy of VGG nets [53] and its effectiveness has been demonstrated in wide applications. Therefore in this work, we focus on DNN repairing at the block level. In particular, we consider both the architecture and weights repairing of a specific block.

2.2 Empirical Study and Motivation

First, we perform a preliminary experiment to discuss the effectiveness of the repairing methods at different levels. In this experiment, we choose 3 variants of ResNet [24] (specifically, ResNet-18,

Table 1. Accuracy (%) and execution time (s/100 epochs) of applying repairing method at different levels on 3 different DNNs trained and tested on CIFAR-10 and Tiny-ImageNet datasets.

	Scale	ResNet-18		ResNet-50		ResNet-101	
		Accuracy (%)	Execution Time	Accuracy (%)	Execution Time	Accuracy (%)	Execution Time
CIFAR-10	**Original**	85.00	-	85.17	-	85.31	-
	Neuron-level	85.18	650.49	85.23	4054.29	85.39	6853.47
	Layer-level	85.16	590.47	85.24	4159.93	85.41	4956.81
	Block-level	85.19	760.94	85.24	3976.39	85.47	7118.03
	Network-level	85.73	1456.92	84.80	5735.61	87.43	9889.35
Tiny-ImageNet	**Original**	45.15	-	46.26	-	46.14	-
	Neuron-level	45.23	1847.59	46.17	13074.85	46.14	20395.79
	Layer-level	45.23	1854.37	46.24	12796.91	46.15	18497.53
	Block-level	45.30	2011.84	46.27	13452.17	46.22	24774.15
	Network-level	45.52	2574.81	46.41	17495.88	46.55	32908.43

ResNet-50, and ResNet-101) as the targeted DNNs ϕ, and we select CIFAR-10 and Tiny-ImageNet dataset as the experimental environment. We repair the DNN at four levels, *i.e.*, Neuron-level (*i.e.*, only fixing weights of one neuron), Layer-level (*i.e.*, only fixing the weights of one layer), Block-level (*i.e.*, fixing the weights of a block) and the Network-level (*i.e.*, fixing all weights of the DNN). Inspired by recent work [55], we choose the neuron (or layer/block) with the greatest gradient (mean gradient for layer and block) as our target to fix. Note that as the previous works have shown that repairing DNN with only a few failure cases is meaningful and important [50, 68], we only randomly select 100 failure cases from the testing dataset to calculate the gradients and choose such neuron (or layer/block). Then, we adjust the weights of the chosen neuron/layer/block by gradient descent w.r.t. the loss function (*e.g.*, cross-entropy loss for image classification). To compare their effectiveness, we apply all methods on the same training dataset of CIFAR-10 and Tiny-ImageNet, then measure the accuracy on the respective testing dataset. We also record the execution time of the total repairing phase (100 epochs) as the indicator of time cost. We show the repairing result in Table 1. Note that we repeat each experiment five times and take the average of each result.

According to Table 1, the network-level repairing achieves the highest accuracy on ResNet-18 and ResNet-101 when repairing on CIFAR-10 dataset, and all 3 variants of ResNet when repairing on Tiny-ImageNet dataset, but also leads to the highest time cost under every configuration. Among 3 other levels of repairing methods, the block-level repairing achieves the highest accuracy improvement without having drastic increment on time cost (*i.e.*, the run-time increment comparing with neuron-level and layer-level is less than 500 seconds on 100 epochs across all 3 ResNets) when repairing on both CIFAR-10 and Tiny-ImageNet.

Overall, the network-level repairing is significantly effective in accuracy improvement but leads to a high time cost. Nevertheless, the block-level repairing achieves impressive accuracy enhancement with much less execution time compared to network-level method (*e.g.*, about 2× less on ResNet-18), making it a good trade-off between effectiveness and efficiency. This fact inspires and motivates us to further investigate the block-level repairing method.

3 BLOCK-LEVEL ARCHITECTURE AND WEIGHTS REPAIRING

In this section, we first provide an overview of our method in the Sec. 3.1 by presenting our intuitive idea and the main pipeline containing two key modules, *i.e.*, *Vulnerable Block Localization* and *Architecture-oriented Search-based Repairing*. After that, we detail the first module in Sec. 3.2 and the second module in Sec. 3.3, respectively. The first module is to locate the vulnerable block in a

deployed DNN, while the second module is to repair the architecture and weights of the localized block by formulating it as an architecture searching problem.

3.1 Overview

Given a deployed DNN $\phi_{(\mathcal{W},\mathcal{A})}$, the weights and architecture usually consist of several blocks, each of which is built by stacking basic operations, *e.g.*, convolutional layer. Then, we represent the weights and architecture with B blocks, *i.e.*, $\mathcal{W} = \{\mathcal{W}_{\mathrm{b}}^i\}_{i=1}^B$ and $\mathcal{A} = \{\mathcal{A}_{\mathrm{b}}^i\}_{i=1}^B$, where the weights or architecture of each block are made up by one or multiple layers. For example, when we consider the ResNet18 [24], we can say that it has six blocks (See Table 2). The first block contains only one convolution layer with the kernel size of $7 \times 7 \times 64$ and the stride of 2. The second to the fifth blocks have two convolutional layers and the last block contains a fully connected layer and a softmax layer. Then, we can reformulate Eq. (1) and Eq. (2) for the proposed block-level repairing by

$$(\mathcal{W}_{\mathrm{b}}^*, \mathcal{A}_{\mathrm{b}}^*) = \mathrm{Locator}(\phi_{(\{\mathcal{W}_{\mathrm{b}}^i\}_{i=1}^B, \{\mathcal{A}_{\mathrm{b}}^i\}_{i=1}^B)}, \mathcal{D}^{\mathrm{repair}}) \tag{3}$$

$$(\hat{\mathcal{W}}_{\mathrm{b}}^*, \hat{\mathcal{A}}_{\mathrm{b}}^*) = \underset{(\mathcal{W}_{\mathrm{b}}^*, \mathcal{A}_{\mathrm{b}}^*)}{\arg\min}\, \mathrm{J}(\phi_{(\mathcal{W}_{\mathrm{b}}^*, \mathcal{A}_{\mathrm{b}}^*)}, \mathcal{D}^{\mathrm{repair}}) \tag{4}$$

where Eq. (3) is to locate the block (*i.e.*, $(\mathcal{W}_{\mathrm{b}}^*, \mathcal{A}_{\mathrm{b}}^*)$) that should be fixed through the proposed adversarial-aware block localization, and Eq. (4) is to repair the localized block by formulating it as a network architecture searching problem. Clearly, compared with the general repairing method (*i.e.*, Eq. (1) and Eq. (2)), the proposed method focuses on fixing the weights and architecture at the block level. We detail the *vulnerable block localization* in Sec. 3.2 and *architecture search-based repairing* in Sec. 3.3.

There are two main solutions for vulnerable neuron localization [13, 55]. The first one employs the neuron spectrum analysis during the forward process of DNN on a testing dataset. It calculates the spectrum of all neurons (*e.g.*, activated/non-activated times of neurons for correctly classified examples and activated/non-activated times of neurons for misclassified examples). These attributes are used to measure the suspiciousness of all neurons. The general principle is that a neuron is more suspicious when the neuron is more often activated under the misclassified examples than that under the correctly classified examples [13]. This solution is able to localize the vulnerable neurons accurately but requires a large testing dataset, which is not suitable for the scenario where a few examples are available for repairing. The second solution is to actively localize the vulnerable neurons by performing backpropagation on the misclassified examples and calculating the gradients of neurons w.r.t. the loss function. The neurons with large gradients are responsible for the misclassification [55]. This solution is able to localize the vulnerable neuron with fewer examples but ignores the effects of correctly classified examples. As shown in Fig. 1, with different failure examples, the gradients of different convolutional blocks in ResNet18 may have similar values, which demonstrates that the gradient-based localization is not sensitive to the variance of the number of failure examples.

Overall, existing methods mainly focus on localizing vulnerable neurons while ignoring the blocks in DNNs. In addition, they have their respective defects. In this work, we propose a novel localization method that aims to find the most vulnerable block in the DNN, which can lead to the buggy behavior of a deployed DNN. To take the respective advantages of existing works and avoid their defects, we propose adversarial-aware spectrum analysis to localize the vulnerable block.

3.2 Adversarial-aware Specturm Analysis for Vulnerable Block Localization

3.2.1 Neuron spectrum analysis. Given a dataset $\mathcal{D}^{\mathrm{repair}}$ for repairing and the targeted DNN $\phi_{(\mathcal{W},\mathcal{A})}$, we calculate the spectrum attributes of the jth neuron in $\mathcal{W}$ by counting the times of activation

Table 2. Network architectures and their respective blocks.

Block	VGGNet 16-layer	ResNet 18-layer	ResNet 50-layer	ResNet 101-layer	EfficientNet B0
Blk1	Conv: $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \\ maxpool \end{bmatrix}$	conv1: 7×7, 64, stride 2			Conv: 3×3, 32, stride 2
Blk2	Conv: $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \\ maxpool \end{bmatrix}$	3×3 max pool, stride 2			MBConv1: 3×3, 16, stride 2
		conv2: $\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	
Blk3	Conv: $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \\ 3\times3, 256 \\ maxpool \end{bmatrix}$	conv3: $\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	MBConv6: 3×3, 24, stride 1
Blk4	Conv: $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \\ 3\times3, 512 \\ maxpool \end{bmatrix}$	conv4: $\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	MBConv6: $\begin{bmatrix} 5\times5, 40 \\ 5\times5, 40 \end{bmatrix}$, stride 2
Blk5	Conv: $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \\ 3\times3, 512 \\ maxpool \end{bmatrix}$	conv5: $\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	MBConv6: $\begin{bmatrix} 3\times3, 80 \\ 3\times3, 80 \\ 3\times3, 80 \end{bmatrix}$, stride 2
Blk6	FC: $\begin{bmatrix} 4096-d \\ 4096-d \\ 1000-d \end{bmatrix}$	average pool, 1,000-d fully-connection, softmax			MBConv6: $\begin{bmatrix} 5\times5, 112 \\ 5\times5, 112 \\ 5\times5, 112 \end{bmatrix}$, stride 1
Blk7	–	–			MBConv6: $\begin{bmatrix} 5\times5, 192 \\ 5\times5, 192 \\ 5\times5, 192 \\ 5\times5, 192 \end{bmatrix}$, stride 2
Blk8	–	–			MBConv6: 3×3, 320, stride 1
Blk9	–	–			1×1 Conv, pool, 1000-d FC

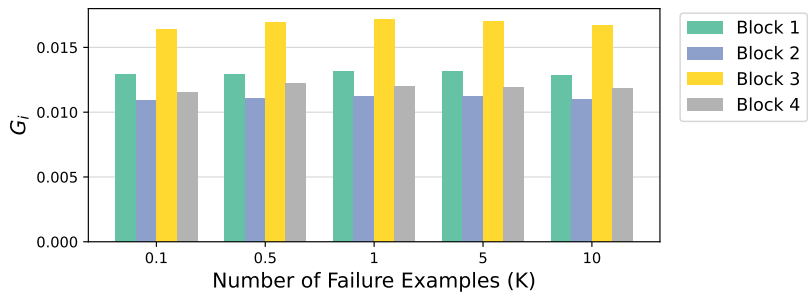


Fig. 1. Average gradients of different blocks in ResNet-18 for different $\mathcal{D}_{\text{fail}}^{\text{repair}}$ sizes.

and non-activation for the neuron under the correctly classified examples and denote them as N_{ac}^{j} and N_{nc}^{j}, respectively. Similarly, we can count the times of activation and non-activation for the same neuron under the misclassified examples and name them as N_{am}^{j} and N_{nm}^{j}, respectively. Then,

we calculate a suspiciousness score for each neuron via the Tarantula measure [28],

$$s_j = \frac{N^j_{\text{am}}/(N^j_{\text{am}} + N^j_{\text{nm}})}{N^j_{\text{am}}/(N^j_{\text{am}} + N^j_{\text{nm}}) + N^j_{\text{ac}}/(N^j_{\text{ac}} + N^j_{\text{nc}})} \tag{5}$$

where s_j determines the suspiciousness of the jth neuron and the higher s_j means the jth neuron is more vulnerable.

3.2.2 Adversarial-aware block spectrum analysis. With the above neuron spectrum analysis, we can obtain the suspiciousness scores for all neurons and the suspiciousness set $\mathcal{S} = \{s_j\}$. Nevertheless, these suspiciousness scores depend on the statistical analysis and are not related to the objective directly, which leads to less effective localization. To alleviate the issue, we propose to refine the suspiciousness scores with adversarial information under the guidance of the loss function (*e.g.*, cross-entropy function for classification).

Specifically, we select the failure examples in $\mathcal{D}^{\text{repair}}$ and construct a subset denoted as $\mathcal{D}^{\text{repair}}_{\text{fail}}$. For each example in $\mathcal{D}^{\text{repair}}_{\text{fail}}$, we can calculate the gradient of all neurons w.r.t. the loss function. Then, we average the gradients of a neuron on all examples and get a set $\mathcal{G} = \{g_j\}$ where g_j is the averaging gradient of the jth neuron on all examples in $\mathcal{D}^{\text{repair}}_{\text{fail}}$. Intuitively, the larger gradient means that the corresponding neuron may significantly contribute to misclassification and should be tuned to minimize the loss. For the ith block, we denote its gradient as the average of the gradients of all neurons in that block, *i.e.*, $G_i = \frac{1}{|\mathcal{W}^i_{\text{b}}|}\sum_{\mathbf{w}_j \in \mathcal{W}^i_{\text{b}}} g_j$. We also calculate the averaging gradient across all neurons, *i.e.*, $\overline{G} = \frac{1}{B}\sum_{i=1}^{B} G_i$. Then, we use these gradients to reweight the suspiciousness scores of all neurons.

$$\hat{s}_j = \frac{|g_j - \overline{G}|}{\max(\{|g_j - \overline{G}|\})} s_j. \tag{6}$$

The principle behind this strategy is that the suspiciousness score of the jth neuron decreases when its relative gradient is small. As a result, we can update the suspiciousness set $\mathcal{S}$ and get $\hat{\mathcal{S}} = \{\hat{s}_j\}$.

A block in the DNN consists of a series of neurons and we collect the updated suspiciousness scores of the neurons in the ith block to the set $\hat{\mathcal{S}}_i \in \hat{\mathcal{S}}$. There are B suspiciousness sets and $\hat{\mathcal{S}} = \{\hat{\mathcal{S}}_i\}_{i=1}^{B}$. After that, we use a threshold (*i.e.*, ϵ) to select the vulnerable neurons, that is, the neuron with $\hat{s}_j > \epsilon$ is identified as the vulnerable neuron. Then, we can count the number of vulnerable neurons in each $\hat{\mathcal{S}}_i$ and the block with the most vulnerable neurons are identified as the targeted block we would repair.

We summarize the whole process of the block localization in Algorithm 1. We first calculate the suspiciousness score of all neurons (Line 1), and calculate the average gradients on each neuron (Line 2). Then, we update the suspiciousness score by calculating the average gradients on each block (Line 3). Finally, we select a threshold to identify the vulnerable blocks (Line 4:5). To validate its advantages, we conduct an experiment to compare the effectiveness and stability of the blocks positioned from $\mathcal{S}$ and $\hat{\mathcal{S}}$, respectively. To compare the stability of the method, we changed the size of the dataset $\mathcal{D}^{\text{repair}}_{\text{fail}}$. We observe that as the size of the dataset changes, the suspicious neurons on each block obtained by $\mathcal{S}$ vary significantly while those obtained by $\hat{\mathcal{S}}$ are much more stable and lead to unanimous conclusions. As shown in Fig. 2, according to the experiments on ResNet-18, by the number of suspicious neurons contained in the block, $\mathcal{S}$ and $\hat{\mathcal{S}}$ estimated that 'block 1' and 'block 4' are the most vulnerable, respectively. We observed similar results when the threshold ϵ are set to other values (*e.g.*, ϵ_{10}, ϵ_{20}, ϵ_{30}, ϵ_{40}, ϵ_{100}). We also conduct detailed quantitative analysis and

discussion in Sec. 5.3, presenting that repairing the most vulnerable block, *i.e.*, 'block 4', achieves much higher improvement.

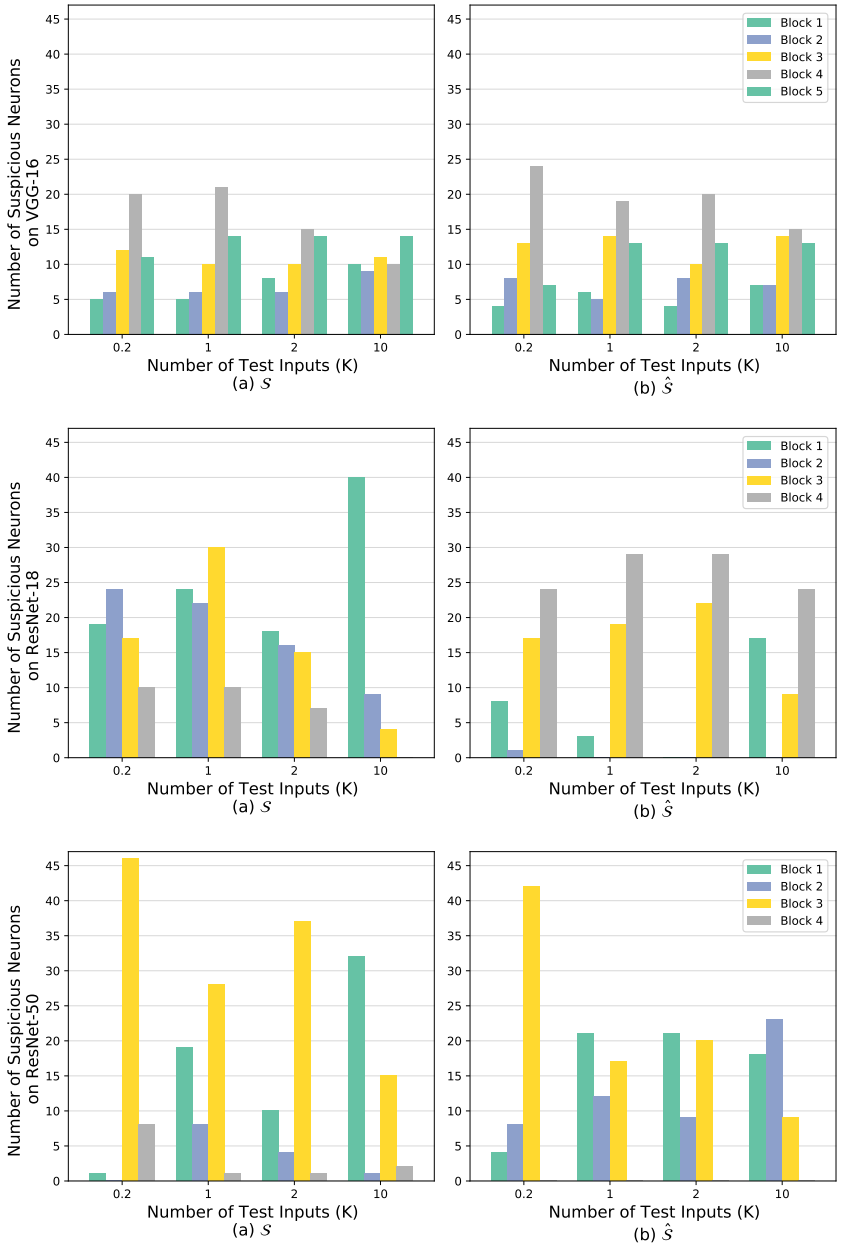


Fig. 2. Collected suspicious neurons in blocks of VGGNet-16, ResNet-18 and ResNet-50 when setting threshold ϵ equal to the value that select top-50 neurons from suspicious ranking, with $\mathcal{S}$(left) and $\hat{\mathcal{S}}$(right), respectively.

Algorithm 1: Vulnerable block localization

Input: A DNN $\phi_{(\mathcal{W},\mathcal{A})}$ and datasets $\mathcal{D}^{\text{repair}}$ and $\mathcal{D}^{\text{repair}}_{\text{fail}}$
Output: $\mathcal{W}^*_{\text{b}}$,$\mathcal{A}^*_{\text{b}}$

1 Calculate suspiciousness scores $\mathcal{S}$ of all neurons via Eq. (5);
2 Calculate the gradients of all neurons on $\mathcal{D}^{\text{repair}}_{\text{fail}}$ and get $\mathcal{G}$;
3 Update the suspiciousness scores $\mathcal{S}$ and get $\hat{\mathcal{S}}$;
4 Identify the vulnerable neurons via a threshold ϵ;
5 Localize the vulnerable block with the maximum number of vulnerable neurons;

3.3 Architecture-oriented Search-based Repairing

After localizing the targeted block, how to break the old architecture's bottleneck and fix it to become competent in the tasks is another challenge. To this end, we formulate the very first block-level architecture and weights repairing as the network architecture search task. Given a deployed DNN with pre-trained weights and fixed architecture (*i.e.*, $\phi_{(\mathcal{W},\mathcal{A})}$), we first relax the targeted block (*i.e.*, $\phi_{(\mathcal{W}^*_{\text{b}},\mathcal{A}^*_{\text{b}})}$) to a directed acyclic graph like the cell structure in the differentiable architecture search (DARTS) [37], which is composed of an ordered sequence of nodes that are connected by edges. Intuitively, the node corresponds to the deep feature while the edge denotes the operation layer like convolutional layer. Our goal is to optimize the edges, *i.e.*, to determine which two nodes should be connected and which operation should be selected for that connection. To this end, the key issues are to define the architecture search space and optimization strategy.

3.3.1 Architecture search space for the targeted block. To better illustrate the process of architecture search, we take ResNet as an example. Given a block in ResNet containing K operation layers, we reformulate it as a directed acyclic graph that has $K+1$ nodes $\{\mathbf{X}^k\}_{k=1}^{K}$ and allow each node to accept the outputs from all previous nodes instead of following the sequential order. As shown in Fig. 3, we present an example of the graph representation of the targeted block via nodes and edges. Specifically, we denote the edge for connecting the ith and jth nodes as $\text{e}_{(i,j)}$ and the node $\mathbf{X}^j$ can be calculated by

$$\mathbf{X}^j = \sum_{i=[1,j-1]} \text{e}_{(i,j)}(\mathbf{X}^i), \tag{7}$$

where $\text{e}_{(i,j)}(\mathbf{X}^i)$ is an edge taking the node $\mathbf{X}^i$ as the input. Then, we define an operation set O containing six candidate operations as presented in Table 3, each of which can be set as the edge. This set of operations is selected in coordination with our NAS method [66]. For example, when we select 'None' for $\text{e}_{(i,j)}$, the two nodes $\mathbf{X}^i$ and $\mathbf{X}^j$ should not be connected.

Note that, the raw sequentially ordered block of ResNet is a special case in the defined search space and we can naturally inherent the raw weights and architecture setup as the initialization for the following optimization.

3.3.2 Architecture and weights optimization. The optimization goal is to select a suitable operation for each edge from the operation set. To this end, we relax the selection as a continuous process by regarding the edge connecting the node i and j as a weighted combination of the outputs of all candidate operations

$$\text{e}_{(i,j)}(\mathbf{X}^i) = \sum_{\text{o}\in O} \frac{\exp\left(\alpha^{\text{o}}_{(i,j)}\right)}{\sum_{\text{o}'\in O}\exp\left(\alpha^{\text{o}'}_{(i,j)}\right)} \text{o}(\mathbf{X}^i) \tag{8}$$

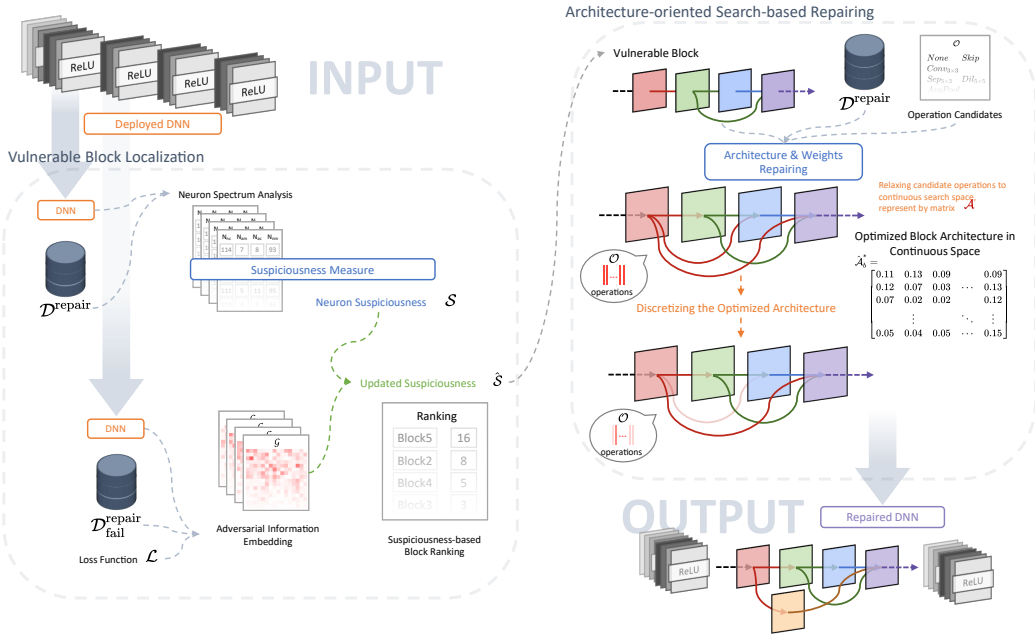


Fig. 3. The overall workflow of *ArchRepair*. Given a deployed DNN model, we first apply the Vulnerable Block Localization to identify the most vulnerable block. Then, we continue to formulate the block repairing as a DNN architecture search problem, and the block's architecture and parameters are optimized jointly through Architecture-oriented Search-based Repairing.

where the parameter $\alpha^{o}_{(i,j)}$ determines the combination weight of using the operation o for connecting the ith and jth nodes. As a result, we can define the architecture parameters for the edge $e_{(i,j)}$ as a vector $\mathbf{a}_{(i,j)} = [\alpha^{o}_{(i,j)} | o \in O]$ assigning each operation in the O a combination weight. Moreover, for the whole block, we denote its architecture as $\mathcal{A}^*_b = \{\mathbf{a}_{(i,j)}\}$ and respective parameters for all candidate operations as $\mathcal{W}^*_b = \{\mathbf{w}_{(i,j)}\}$. Then, we can specify the repairing process in Eq. (4) by optimizing the weights (*i.e.*, $\mathcal{W}^*_b$) and architecture parameters (*i.e.*, $\mathcal{A}^*_b$) on the training dataset and validation dataset, alternatively, that is, we have

$$\hat{\mathcal{W}}^*_b = \arg\min_{\mathcal{W}^*_b} J(\phi_{(\mathcal{W}^*_b, \mathcal{A}^*_b)}, \mathcal{D}^{\text{repair}}_{\text{train}}), \tag{9}$$

$$\hat{\mathcal{A}}^*_b = \arg\min_{\mathcal{A}^*_b} J(\phi_{(\hat{\mathcal{W}}^*_b, \mathcal{A}^*_b)}, \mathcal{D}^{\text{repair}}_{\text{val}}) \tag{10}$$

where $J(\cdot)$ is specified as the cross-entropy loss function for the image classification task. During the training process, we initialize the block architecture $\mathcal{A}^*_b$ as the raw block architecture of the targeted DNN, and update the architecture and weights, alternatively. We will illustrate the repairing process in Sec. 3.4. After getting the optimized architecture (*i.e.*, $\hat{\mathcal{A}}^*_b$) in the continuous search space, we set the operation with maximum combination weight as the edge, *i.e.*, $e_{(i,j)} = \arg\max_{o \in O} \alpha^{o}_{(i,j)}$. Then, we retrain the weights $\hat{\mathcal{W}}^*_b$ with fixed block architecture.

Table 3. All operators in the operation set O.

Operators	Operations
None	Add a Zero CNN layer whose weights are all zero.
Skip	Add an Identity CNN layer whose weights are all one.
AvgPool	Add an Average Pooling layer and an Identity CNN layer.
MaxPool	Add a Max Pooling layer and an Identity CNN layer.
SepConv	Add separated CNN layers.
DilConv	Add a CNN layer with the dilation kernel and an Identity CNN layer.

3.4 Our Repairing Algorithm of ArchRepair

Fig. 3 summarizes the whole workflow of *ArchRepair*. Given a deployed DNN, we first employ the proposed vulnerable block localization to determine the block we aim to repair. Specifically, we use the repair dataset $\mathcal{D}^{\text{repair}}$ and the neuron spectrum analysis to calculate the suspiciousness of all neurons, *i.e.*, $\mathcal{S} = \{s_j\}$. Meanwhile, we use the failure examples in $\mathcal{D}^{\text{repair}}$ (*i.e.*, $\mathcal{D}^{\text{repair}}_{\text{fail}}$) to obtain the gradients of all neurons w.r.t. the loss function (*i.e.*, $\mathcal{G} = \{g_j\}$). Then, we use Eq. (6) and the gradients $\mathcal{G} = \{g_j\}$ to reweight $\mathcal{S} = \{s_j\}$, thus get the suspiciousness scores $\hat{\mathcal{S}} = \{\hat{s}_j\}$. After that, we can calculate the number of vulnerable neurons through a threshold ϵ, that is, when the suspiciousness score of a neuron $\hat{\mathcal{S}} = \{\hat{s}_j\}$ is larger than ϵ, the neuron is identified as a vulnerable case. Finally, the block with the largest number of vulnerable cases is selected as the targeted block we want to repair.

During the architecture search-based repairing, we reformulate the targeted block as a directed acyclic graph, where the deep features are nodes and operations are edges. Then, we relax each edge as a combination of six operations (*i.e.*, Eq. (8)), where the combination weights correspond to the architecture parameters $\mathcal{A}^*_{\text{b}} = \{\mathbf{a}_{(i,j)}\}$. We use the dataset $\mathcal{D}^{\text{repair}}$ to conduct the architecture and weights optimization via Eq. (9) and Eq. (10), where the original architecture and weights are inherited and serve as the optimization initialization. Therefore, given the optimized block architecture in the continuous space (*i.e.*, $\hat{\mathcal{A}}^*_{\text{b}}$), we discretize it to the final architecture by preserving the operation with the maximum combination weight and removing other operations. Finally, we use the $\mathcal{D}^{\text{repair}}$ to fine-tune the weights by fixing the optimized architecture for the repaired DNN.

4 EXPERIMENTAL DESIGN AND SETTINGS

In this section, we conduct extensive experiments to validate the proposed methods and compare with the state-of-the-art DNN repair techniques, to investigate the following research questions:

- **RQ1.** Does *ArchRepair* outperform the state-of-the-art (SOTA) DNN repair techniques with better repairing effects?
- **RQ2.** Could *ArchRepair* repair DNNs on certain failure patterns without sacrificing robustness on clean data and other failure patterns?
- **RQ3.** Is our proposed localization method effective in identifying vulnerable neuron blocks?
- **RQ4.** How do different components of our proposed method impact the overall repairing performance?

RQ1 intends to evaluate the overall repairing capability of *ArchRepair* and to compare it to SOTA DNN repair techniques as baselines. **RQ2** aims at exploring the potential of our method in repairing DNN on corrupted data, which are common robustness issues during DNN practical usage in the operational environments. **RQ3** intends to examine whether the proposed localization method can

precisely locate vulnerable blocks. **RQ4** is to explore the contribution that each of *ArchRepair*'s key components makes on the overall performance of DNN repair.

4.1 Experimental Setups

To answer the research questions above, we design our evaluation from multiple perspectives listed in the following.

Subject Datasets and Repairing Scenarios. Given a deployed DNN trained on a training dataset $\mathcal{D}^{\mathrm{t}}$, we can evaluate it on a testing dataset $\mathcal{D}^{\mathrm{v}}$. In the real world, there are a lot of scenes that cannot be covered by $\mathcal{D}^{\mathrm{v}}$ and the DNN's performance may decrease significantly after the DNN is deployed in its operational environment. For example, there are common corruptions (*i.e.*, noise patterns) in the real world that can affect the DNN significantly [25]: Gaussian noise (GN), shot noise (SN), impulse noise (IN), defocus blur (DB), Gaussian blur (GB), motion blur (MB), zoom blur (ZB), snow (SNW), frost (FRO), fog (FOG), brightness (BR), contrast (CTR), elastic transform (ET), pixelate (PIX), and JPEG compression (JPEG).

According to the aftermentioned situations, we consider two repairing scenarios that commonly occur in practice:

- **Repairing the accuracy drift on testing dataset.** When we evaluate the DNN on the testing dataset $\mathcal{D}^{\mathrm{v}}$, we can collect a few failure examples (*i.e.*, 1,000 examples) denoted as $\mathcal{D}^{\mathrm{v}}_{\mathrm{fail}}$. Then, we set $\mathcal{D}^{\mathrm{repair}} = \mathcal{D}^{\mathrm{v}}_{\mathrm{fail}} \cup \mathcal{D}^{\mathrm{t}}$ and use the proposed or baseline repairing methods to enhance the deployed DNNs. We evaluate the accuracy on the testing dataset where $\mathcal{D}^{\mathrm{v}}_{\mathrm{fail}}$ is excluded (*i.e.*, $\mathcal{D}^{\mathrm{v}} \setminus \mathcal{D}^{\mathrm{v}}_{\mathrm{fail}}$). Note that, the context of repairing DNN with only a few testing data is meaningful and important, which is adopted by recent works [50, 68]. In addition, there could be many practical scenarios, where collecting buggy examples is very difficult or at very high costs, with only a few buggy examples collected entirely. Hence, we follow the common choice in recent works [50, 68] to select only 1,000 failure examples from testing data.
- **Repairing the robustness on corrupted datasets.** When we evaluate the DNN on a corrupted testing dataset $\mathcal{D}^{\mathrm{c}}$, we can also collect a few failure examples (*i.e.*, 1,000 examples) denoted as $\mathcal{D}^{\mathrm{c}}_{\mathrm{fail}}$ and set $\mathcal{D}^{\mathrm{repair}} = \mathcal{D}^{\mathrm{c}}_{\mathrm{fail}} \cup \mathcal{D}^{\mathrm{t}}$. The repairing goal is to enhance the accuracy on $\mathcal{D}^{\mathrm{c}} \setminus \mathcal{D}^{\mathrm{c}}_{\mathrm{fail}}$ and other corrupted datasets while maintaining the accuracy on the clean testing dataset (*i.e.*, $\mathcal{D}^{\mathrm{v}} \setminus \mathcal{D}^{\mathrm{v}}_{\mathrm{fail}}$).

We choose CIFAR-10 [31], CIFAR-100 [31], Tiny-ImageNet [34] and ImageNet [10] as the evaluation datasets. They are commonly used datasets in recent DNN repair studies, enabling us to perform comparative studies in a relatively fair way. Each dataset contains its respective training dataset $\mathcal{D}^{\mathrm{t}}$ and testing dataset $\mathcal{D}^{\mathrm{v}}$. CIFAR-10 contains a total of 60,000 images in 10 categories, in which 50,000 images are for $\mathcal{D}^{\mathrm{t}}$ and the other 10,000 are for $\mathcal{D}^{\mathrm{v}}$. CIFAR-100 has 100 classes containing 600 images each. There are 500 images in the training dataset $\mathcal{D}^{\mathbf{t}}$ and 100 images in the testing dataset $\mathcal{D}^{\mathbf{v}}$ for each class. Tiny-ImageNet has a training dataset $\mathcal{D}^{\mathrm{t}}$ with the size of 100,000 images, and a testing dataset $\mathcal{D}^{\mathrm{v}}$ with the size of 10,000 images. ImageNet contains over 14 million images. In our experiment, the training dataset $\mathcal{D}^{\mathrm{t}}$ uses 1.3 million images, and the testing dataset $\mathcal{D}^{\mathrm{v}}$ uses over 50000 images. Therefore, we have corrupted testing datasets $\{\mathcal{D}^{\mathrm{c}}_i\}$ where $i = 1, 2, \ldots, 15$ corresponding to the above fifteen corruptions [25].

DNN architectures. We select six different architectures of DNN, *i.e.*, VGGNet-16 [54], ResNet-18, ResNet-50, ResNet-101 [24], DenseNet-121 [26] and EfficientNet-B0 [57]. Given that *ArchRepair* is a block-based repairing method, the block-like architecture, ResNet, turns out to be a perfect

research subject. For a broad comparison, we also choose a non-block-like architecture, DenseNet-121, to examine the repairing capability of *ArchRepair* [1]. For each architecture, we first pre-train them with the original training dataset $\mathcal{D}^{\mathrm{t}}$ (from CIFAR-10, CIFAR-100, Tiny-ImageNet or ImageNet), the model with the highest accuracy in testing dataset $\mathcal{D}^{\mathrm{v}}$ (from CIFAR-10, CIFAR-100, Tiny-ImageNet or ImageNet) will be saved as pre-trained model ϕ_θ. As the original ResNet and DenseNet are not designed for CIFAR-10 and Tiny-ImageNet datasets, we use the unofficial architecture code offered by a popular GitHub project[2], which has more than 4.1K stars.

Block definition. We divide each of the six selected DNN architectures into several blocks. For each of ResNet-18, ResNet-50, and ResNet-101, we follow its block structures and divide it into four blocks, as shown in Table 2. For DenseNet-121, we divide it by every two convolutional layers as one block. For VGGNet-16, we manually divide it into six blocks by maxpool layer as Table-2 shows and select five of them as repairing targets (*i.e.*, Block 1 ~5, Block 6 is used for getting output, so we left it out of repairing). For EfficientNet-B0, we follow its block structures and divide them into seven blocks (see Table 2).

NAS method. We select PC-DARTS [66] as our NAS solution for *ArchRepair*. While all popular NAS techniques should fit into *ArchRepair*(*e.g.*, DARTS, SNAS, and BayesNAS), these techniques use more time than PC-DARTS in searching for a better network architecture. Given that DNN repair is a time-sensitive task, we choose PC-DARTS [66] as it is among the fastest NAS methods. Though *ArchRepair*remains the interface for switching to other NAS techniques when the task cares more about the performance of repaired models than the time cost of repairing.

Hyper-parameters. Regarding the training setup, we employ stochastic gradient descent (SGD) as the optimizer, setting batch size as 128, the initial learning rate as 0.1, and the weight decay as 0.0005. We use cross-entropy loss as the loss function. The maximum number of epochs is 500, and an early-stop function will terminate the training phase when the validation loss no longer decreases in 10 epochs.

Baselines. To demonstrate the repairing capability of the proposed *ArchRepair*, we select six SOTA DNN repair methods from two different categories as baselines: neuron-level repairing methods and network-level repairing methods. The neuron-level repairing methods focus on fixing certain neurons' weights in order to repair the DNNs. Representative methods from this category are MODE [43], Apricot [70], and Arachne [55]. While network-level repairing methods mainly repair DNNs by using augmented datasets to fine-tune the whole network, where SENSEI [18], Few-Shot [50], and DeepRepair [68] are the most popular ones. For a fair comparison, we employ the same settings on all six repairing methods and *ArchRepair*. In order to fully evaluate the effectiveness of the proposed method, we apply all methods (six baselines and *ArchRepair*) to fix four different DNN architectures on large-scale datasets, including the clean version and 15 corrupted versions from CIFAR-10 and Tiny-ImageNet, to assess the repairing capability.

Other configurations. We implement *ArchRepair* in Python 3.9 based on PyTorch framework. All the experiments were performed on a server with a 12-core 3.60GHz Xeon CPU E5-1650, 128GB RAM and four NVIDIA GeForce RTX 3090 GPUs (each has 4GB memory), which runs Ubuntu 18.04.

In summary, for each baseline method and *ArchRepair*, our evaluation consists of 96 configurations (6 DNN architectures × 16 versions of a dataset [3]) on four datasets (*i.e.*, CIFAR-10, CIFAR-100, Tiny-InageNet and ImageNet. For CIFAR-10 dataset, an execution of training and repairing a model under one specific configuration costs about 12 hours on average (the maximum one is about 50 hours); while for Tiny-ImageNet dataset, an execution of training and repairing a model takes

[1]For DenseNet-121, we manually group two consecutive convolution blocks as one block when repairing.

[2]Train CIFAR10 with PyTorch: https://github.com/kuangliu/pytorch-cifar

[3]one clean dataset (repairing the accuracy drift on testing dataset) and fifteen corruption datasets (repairing the robustness on corrupted datasets)

about 18 hours on average (the maximum one is about 64 hours). We measured the execution time of repairing six different DNN architectures (*i.e.*, VGGNet-16, ResNet-18, ResNet-50, ResNet-101, DenseNet-121, and EfficientNet-B0) repaired on CIFAR-10 within 100 epochs by different repairing methods. The results are reported in Table 4. According to Table 4, the Neuron-lv's methods use less execution time than other repairing methods (The cell with green background), and the Network-lv's methods use more execution time than others. Our method, *ArchRepair*, uses more execution time than Neuron-lv's methods but less than Network-lv's methods. This is because *ArchRepair* repairs DNN models on the block level, which works at a larger size than the neuron level but at a smaller size than the network level. This is also consistent with our expectation in Sec. 2.2, *i.e.*, block-level repairing makes a good trade-off between effectiveness and efficiency. Overall, the total execution time of our experiments takes more than two months.

Table 4. Execution time of repairing 6 different DNNS (*i.e.*, VGGNet-16, ResNet-18, ResNet-50, ResNet-101, DenseNet-121, and EfficientNet-B0) repaired on CIFAR-10 within 100 epochs by different repairing methods. The result describes ArchRepair uses less repairing time than Network-lv's methods and has excellent repairing performance.

	CIFAR-10	Execution time (100 epochs)					
		VGG-16	**Res-18**	**Res-50**	**Res-101**	**Den-121**	**Eff-B0**
Neuron-lv	**MODE [43]**	2h17m	2h41m	3h37m	4h36m	6h58m	4h11m
	Apricot [70]	3h09m	3h31m	4h15m	5h32m	8h06m	4h15m
	Arachne [56]	2h17m	2h31m	3h54m	4h58m	7h25m	3h09m
Network-lv	**SENSEI [18]**	46h18m	48h21m	54h25m	68h32m	82h36m	57h03m
	Few-Shot [51]	27h18m	28h51m	32h11m	40h25m	47h36m	30h39m
	DeepRepair [68]	49h21m	51h47m	56h23m	65h23m	72h16m	61h32m
ArchRepair (ours)		18h15m	18h37m	21h34m	29h17m	33h25m	23h17m

5 EXPERIMENTAL RESULTS

In this section, we summarize the high-level results and findings to answer our research questions. We present more detailed evaluation results, configurations as well as a replication package on our supplementary website [56] of this paper.

5.1 RQ1: Does *ArchRepair* outperform the state-of-the-arts (SOTA) DNN repair techniques?

To answer RQ1, we train 6 DNNs (*i.e.*, VGGNet-16, ResNet-18, ResNet-50, ResNet-101, DenseNet-121 and EfficientNet-B0) on 4 datasets' (*i.e.*, CIFAR-10, CIFAR-100, Tiny-ImageNet and ImageNet) training datasets (*i.e.*, $\mathcal{D}^{\mathrm{t}}$) and evaluate them on testing datasets (*i.e.*, $\mathcal{D}^{\mathrm{v}}$) respectively. To evaluate the performance of our method (*i.e.*, *ArchRepair*), we apply six different SOTA methods as well as *ArchRepair* to repair these 4 DNNs. The evaluation results of repairing are summarized in Table 5. In general, *ArchRepair* exhibits significant advantages over all baseline methods on the 6 DNNs, demonstrating the effectiveness and generalization ability of the proposed method. In particular, comparing with the state-of-the-art DNN repair methods (*i.e.*, neuron-level repairing method Arachne [55], and network-level repairing method DeepRepair [68]), *ArchRepair* achieves much higher accuracy on 5 out of 6 DNNs on CIFAR-10 dataset. On the more challenging dataset, Tiny-ImageNet, *ArchRepair* still achieves much higher accuracy on 3 out of 6 DNNs. Note that on DenseNet-121, all the repairing methods failed to repair, *i.e.*, failing to improve the performance compared to the original network. One possible explanation is that the original DenseNet-121's performance has almost reached the upper bound of the classification accuracy on Tiny-ImageNet,

Table 5. Average accuracy (%) of 6 different DNNs (*i.e.*, VGGNet-16, ResNet-18. ResNet-50, ResNet-101, DenseNet-121, and EfficientNet-B0) repaired on 4 dataset (*i.e.*, CIFAR-10, Tiny-ImageNet, CIFAR-100, and ImageNet) by different repairing methods. Note that each configuration of the experiment is repeated 5 times and the average results are summarized.

Baseline	CIFAR-10						Tiny-ImageNet					
	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0
Original	83.72	85.00	85.17	85.72	87.97	88.93	42.87	45.15	46.27	46.14	48.73	49.25
MODE [43]	84.22	85.48	85.08	85.78	88.64	88.89	43.37	45.47	45.90	46.40	47.86	51.61
Apricot [70]	84.33	86.78	88.95	89.25	90.23	86.97	43.04	46.08	46.79	45.35	45.14	50.48
Arachne [55]	84.23	85.09	87.45	89.35	91.20	88.99	44.17	46.93	47.14	46.43	46.93	51.60
SENSEI [18]	84.53	85.07	86.33	89.13	89.45	88.89	44.49	45.91	47.10	46.12	45.70	52.40
Few-Shot [50]	84.67	86.21	86.49	88.06	88.44	88.14	44.01	46.32	46.62	45.90	45.31	52.42
DeepRepair [68]	85.00	86.68	87.08	88.98	90.78	92.12	45.84	46.92	47.69	46.58	46.68	52.45
ArchRepair (ours)	85.58	88.53	88.96	90.20	91.36	91.35	45.69	46.96	47.51	46.75	46.37	52.62

Baseline	CIFAR-100						ImageNet					
	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0
Original	60.27	62.58	63.74	63.97	65.89	67.42	64.72	64.81	65.79	66.28	67.85	69.05
MODE [43]	61.08	63.76	63.87	65.04	65.10	67.93	64.29	64.95	65.59	67.32	68.29	69.54
Apricot [70]	62.89	63.48	65.19	64.03	66.75	68.21	65.39	65.53	65.64	67.38	67.58	69.85
Arachne [55]	62.91	63.63	65.21	64.60	66.28	67.46	65.10	66.07	66.53	67.06	67.76	70.61
SENSEI [18]	61.96	64.27	65.20	65.64	66.50	69.28	65.37	65.59	66.02	67.62	68.20	71.69
Few-Shot [50]	62.46	63.95	64.92	65.06	67.87	68.69	66.77	66.49	65.72	67.52	68.99	70.34
DeepRepair [68]	62.67	64.36	65.77	66.63	67.95	69.76	66.61	66.41	66.71	67.61	69.91	72.12
ArchRepair (ours)	66.24	65.41	65.67	66.93	67.02	68.21	66.96	67.10	67.93	67.88	70.09	70.79

hence there might not be much room for improvement in terms of accuracy. To better illustrate the performance of *ArchRepair* compared with other baselines, we also conduct the statistical test (*i.e.*, *Wilcoxon Signed-rank Test*) on the results obtained by our method, compared with each of the 6 corresponding repairing methods across all 6 different models (*i.e.*, VGGNet-16, ResNet-18/50/101, DenseNet-101 and EfficientNet-B0), all the 4 evaluated datasets (*i.e.*, CIFAR-10/100, Tiny-ImageNet and ImageNet). Table 6 summarizes the obtained statistical test results, which demonstrate the advantage of our method to be statistically significant at the 0.01 confidence level (*i.e.*, $p < 0.01$), compared with the SOTA. We report the obtained significant test results in Table 6 and add a paragraph of discussion in the original paper, the results confirm the advantage of our method to be statistically significant at the 0.01 confidence level (*i.e.*, $p < 0.01$).

Table 6. Wilcoxon signed-rank test

n=120	ArchRepair					
	MODE [43]	Apricot [70]	Arachne [55]	SENSEI [18]	Few-Shot [50]	DeepRepair [68]
p	3.91E-19 < 0.01	5.09E-20 <0.01	1.51E-18 <0.01	3.37E-16 <0.01	5.96E-17 <0.01	1.55E-3 <0.01

Furthermore, to understand the influence of repairing on DNN's robustness, we evaluate the repaired DNNs' performance on corruption datasets (*i.e.*, CIFAR-10-C [25] and Tiny-ImageNet-C [25]). The CIFAR-10-C and Tiny-ImageNet-C contain over 15 types of natural corruption datasets, and we show the results on CIFAR-10-C in Fig. 4 and Tiny-ImageNet-C in Fig. 5. Obviously in Fig. 4, *ArchRepair* achieves the highest accuracy on a majority of corruption datasets across three variants of ResNet (8/15, 9/15, and 7/15 on ResNet-18, ResNet-50, and ResNet-101, respectively) besides the best performance on the clean dataset. Even on DenseNet-121, which is not a block-like DNN, *ArchRepair* also achieves promising performance compared with SOTA method Apricot [70]. The performance of *ArchRepair* is also significant on Tiny-ImageNet-C. As we've mentioned before,

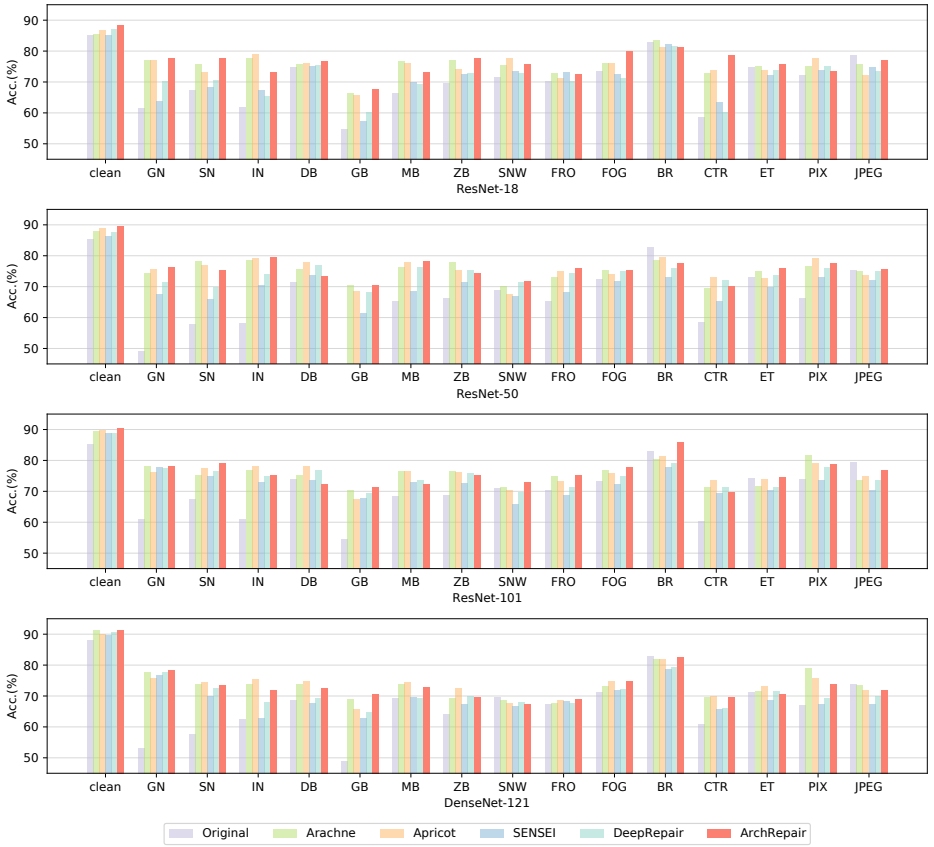


Fig. 4. Comparing the repairing methods on different DNNs (*i.e.*, ResNet-18, ResNet-50, ResNet-101 and DenseNet-121) by contrasting the accuracy of repaired DNNs on CIFAR-10's testing dataset (*i.e.*, $\mathcal{D}^{t}$) and corruption datasets (*i.e.*, $\mathcal{D}^{c}$).

Tiny-ImageNet is way more challenging. Nevertheless, *ArchRepair* still outperforms baselines in terms of the robustness on a majority of corruption datasets across three variants of ResNet (9/15, 9/15, and 7/15 on ResNet-18, ResNet-50, and ResNet-101, respectively) as well as the non-block-like DNN DenseNet-121 (8/15). The results confirm that *ArchRepair* doesn't harm the DNN's robustness, and on the contrary, it can even sometimes improve DNN's generalization ability towards classifying corrupted data.

Answer to RQ1: According to the experimental results on clean dataset, *ArchRepair* outperforms the SOTA repairing method on all 6 DNNs with different architectures (*i.e.*, VGGNet-16, ResNet-18, ResNet-50, ResNet-101, DenseNet-121, and EfficientNet-B0). Moreover, the experimental results on corruption datasets also support that *ArchRepair* can repair a DNN without harming its robustness.

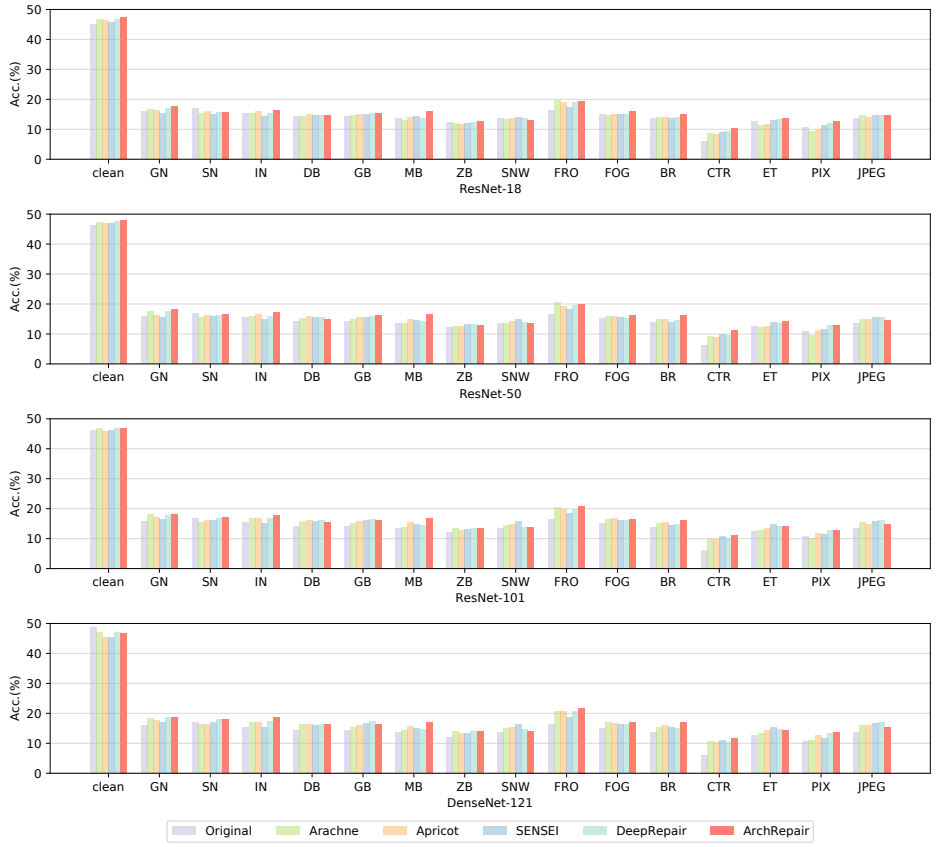


Fig. 5. Comparing the repairing methods on different DNNs (*i.e.*, ResNet-18, ResNet-50, ResNet-101 and DenseNet-121) by contrasting the accuracy of repaired DNNs on Tiny-Imagenet's testing dataset (*i.e.*, $\mathcal{D}^{t}$) and corruption datasets (*i.e.*, $\mathcal{D}^{c}$).

Table 7. Accuracy (%) of a deployed ResNet-18 repaired by different repairing methods on 15 different corruption patterns.

	ResNet-18	Clean	GN	SN	IN	DB	GB	MB	ZB	SNW	FRO	FOG	BR	CTR	ET	PIX	JPEG
CIFAR-10-C	**Original**	85.000	61.452	67.392	61.944	74.762	54.782	66.348	69.476	71.408	70.114	73.532	82.736	58.716	74.822	72.364	78.752
	Apricot [70]	86.644	76.930	78.656	77.694	75.827	66.390	76.810	79.851	76.406	77.269	78.979	89.254	74.390	75.112	75.350	75.810
	Arachne [55]	88.451	77.144	77.715	78.976	76.546	65.815	75.963	77.712	77.862	77.224	79.200	86.913	75.792	73.876	77.694	74.402
	SENSEI [18]	86.525	68.762	70.471	73.345	76.842	60.244	71.229	73.297	73.732	73.814	76.975	83.006	64.861	72.814	75.833	79.495
	DeepRepair [68]	88.159	75.197	73.990	75.807	77.369	63.263	75.703	74.973	76.999	76.872	77.884	83.967	72.889	76.594	74.669	77.726
	***ArchRepair* (ours)**	90.177	77.546	77.689	73.237	80.679	67.523	75.998	77.697	77.867	80.677	79.854	85.146	79.026	78.053	77.448	77.967
Tiny-ImageNet-C	**Original**	45.150	15.912	16.972	15.482	14.281	14.337	13.648	12.191	13.562	16.452	15.119	13.823	6.130	12.657	10.819	13.577
	Apricot [70]	46.732	16.703	15.270	15.339	14.266	14.762	13.047	11.959	13.319	19.550	14.838	14.041	8.790	11.231	9.227	14.825
	Arachne [55]	46.297	16.302	15.932	15.932	14.938	15.152	14.119	11.695	13.805	18.986	15.106	14.123	8.253	11.831	10.145	13.918
	SENSEI [18]	45.824	15.270	14.870	14.390	14.664	15.052	14.191	12.112	13.917	17.250	14.943	13.602	9.117	12.902	11.277	14.772
	DeepRepair [68]	46.780	17.032	15.673	15.277	14.669	15.324	13.570	12.478	13.624	18.950	15.152	14.145	9.385	13.496	11.926	14.597
	***ArchRepair* (ours)**	47.350	17.820	15.779	16.376	14.769	15.224	15.967	12.670	12.923	19.295	15.915	15.112	10.337	13.765	12.553	14.624

5.2 RQ2: Can *ArchRepair* fix DNN on a certain failure pattern without sacrificing robustness on clean data and other failure patterns?

In Sec. 5.1, our investigation results demonstrated that *ArchRepair* will not affect DNN's robustness when repairing on the clean dataset. Hence in this section, we continue to validate whether our method harms DNN's robustness when repairing a specific failure pattern.

We first verify the repairing capability of *ArchRepair*. We repair a deployed DNN (*i.e.*, ResNet-18 [4]) on each of the corruption datasets from CIFAR-10-C and Tiny-ImageNet-C, and compare the performance with the other repairing methods, where the results are summarized in Table 7. Comparing the experimental results on the corruption dataset, we see that all repairing methods have the capability to repair the failure patterns, except shot noise (SN) on Tiny-ImageNet-C (all repairing methods fail to repair this corruption pattern). Among these repairing techniques, our method *ArchRepair* has the highest accuracy on 8 out of 15 the corruption datasets on CIFAR-10-C dataset, and 9 out of 15 the corruption datasets on Tiny-ImageNet-C, respectively, demonstrating that *ArchRepair* exhibits the advantages in repairing failure patterns.

To validate whether our method has harmed DNN's robustness, we also evaluate the performance of repaired DNNs on the other corruption datasets. The evaluation results on CIFAR-10 and Tiny-ImageNet are shown in Fig. 6 and Fig. 7, respectively. Besides, we calculate the robustness of repaired models with the formula used in SENSEI. The results of robustness are recorded in Table 8. Comparing the accuracy difference on CIFAR-10-C (see Fig. 6), we observe that the DNNs repaired by *ArchRepair* (*i.e.*, the red bar) have higher accuracies on both clean and corruption datasets than the original DNN (*i.e.*, the gray bar, which is lower than others in most of the cases), indicating that repairing method will not harm the DNN's robustness when having fixed certain corruption patterns. Also, this fact proves that the repairing procedure will not cause over-fit. This is also verified by the results on Tiny-ImageNet-C (see Fig. 7), where repairing on a certain corruption pattern does not affect the DNN's robustness on clean dataset and other corruption patterns. Instead, it can even enhance the robustness in some cases (*e.g.*, when repairing on Fog corruption is performed, the performance on other corruptions is also improved).

Answer to RQ2: *ArchRepair* can successfully fix a certain corruption pattern on a deployed DNN (*i.e.*, ResNet-18), outperforming the existing 4 DNN repair methods. In addition, *ArchRepair*'s repairing doesn't harm DNN's robustness on clean dataset and other failure patterns.

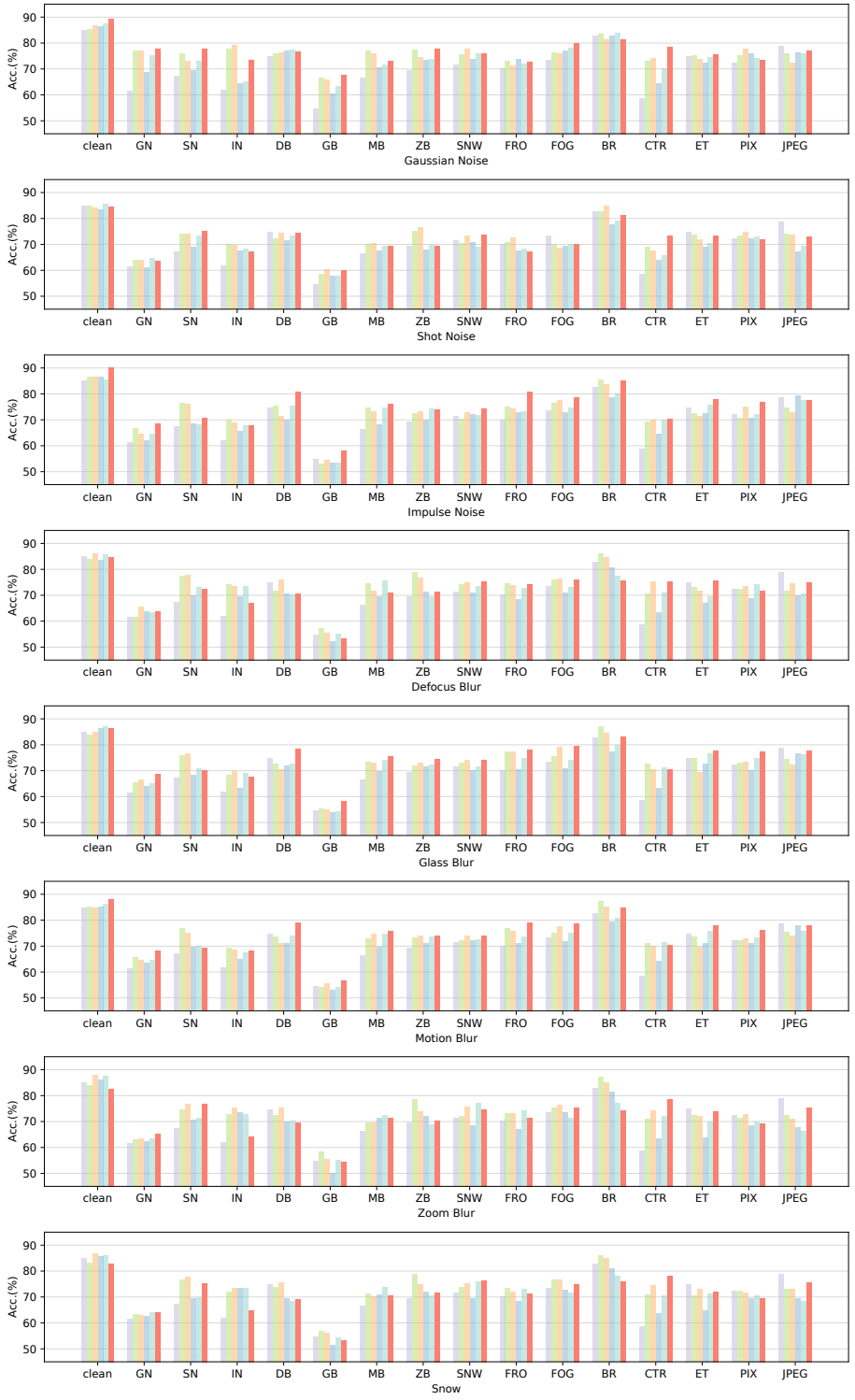
5.3 RQ3: Is our proposed localization effective in identifying vulnerable block candidates?

To verify the effectiveness of our localization method, we conduct an experiment by applying the repairing method on all 4 blocks of ResNet-18 & ResNet-50, and comparing the accuracy on the clean datasets $\mathcal{D}^{\mathrm{v}}$ of both CIFAR-10 and Tiny-ImageNet with their block suspiciousness $\mathcal{S}_{\mathrm{B}}$ (*i.e.*, the number of suspicious neurons in correspond block). We calculate the block suspiciousness under 8 different thresholds ϵ_i [5] ($i \in \{10, 20, 30, 40, 50, 75, 100, 150\}$) to evaluate how the threshold ϵ_i affects the block suspiciousness. The experimental results are summarized in Table 9.

As shown in Table 9, the block suspiciousness $\mathcal{S}_{\mathrm{B}}$ of Block 4 in ResNet-18 and Block 3 in ResNet-50 are always the highest on both CIFAR-10 and Tiny-ImageNet datasets, no matter what value the threshold ϵ_i is. It matches the performance of repaired DNNs, where the DNN repaired on Block 4 in ResNet-18 and Block 3 in ResNet-50 has the highest accuracy, respectively. This demonstrates that our localization method can correctly locate the most vulnerable block.

[4] More evaluation results on other DNNs are available on our project website [56]

[5] ϵ_i indicates top-i neurons with highest suspiciousness.


Acc.(%)
90
80
70
60
50
clean
GN
SN
IN
DB
GB
MB
ZB
SNW
FRO
FOG
BR
CTR
ET
PIX
JPEG
Gaussian Noise
Shot Noise
Impulse Noise
Defocus Blur
Glass Blur
Motion Blur
Zoom Blur
Snow

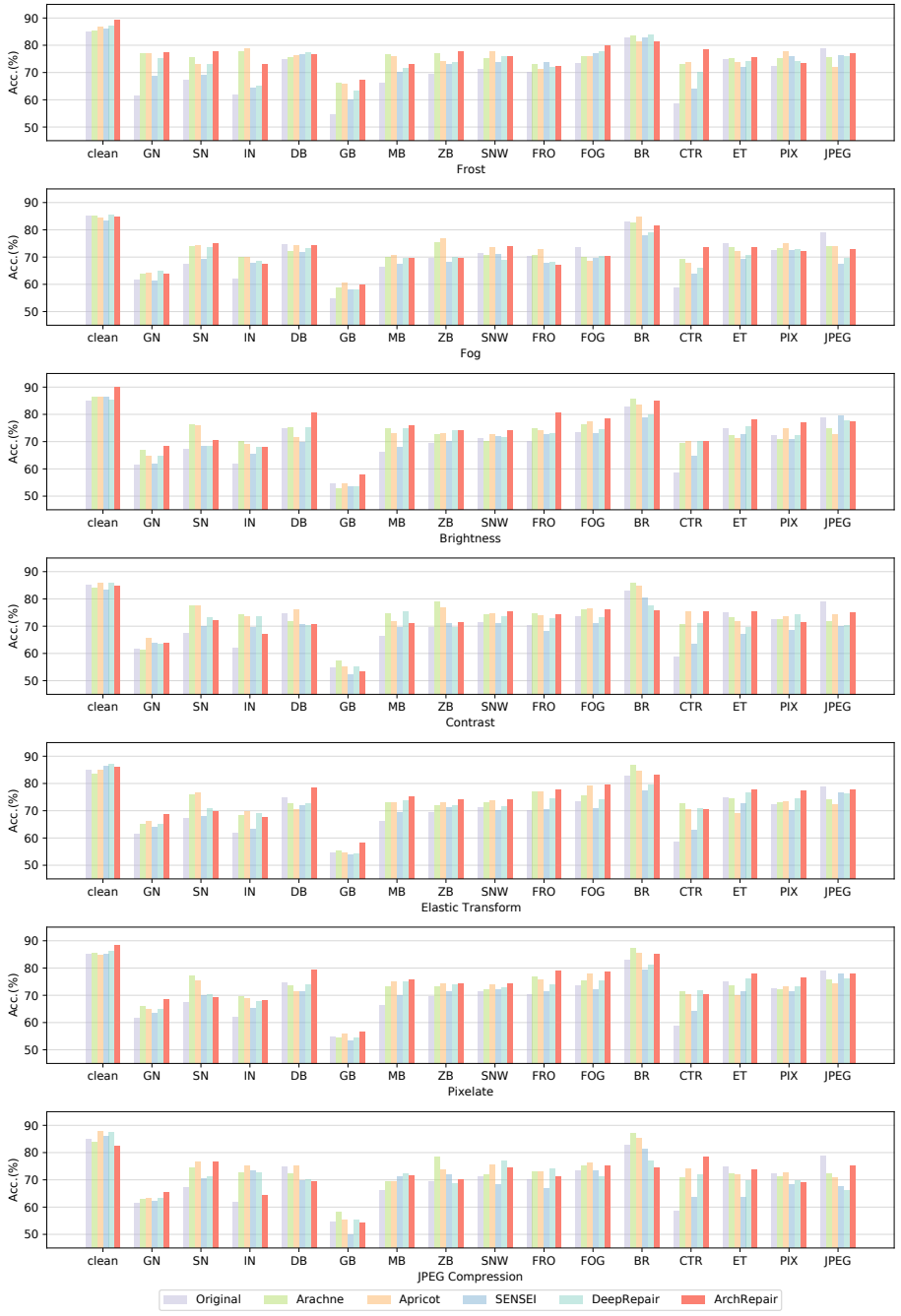
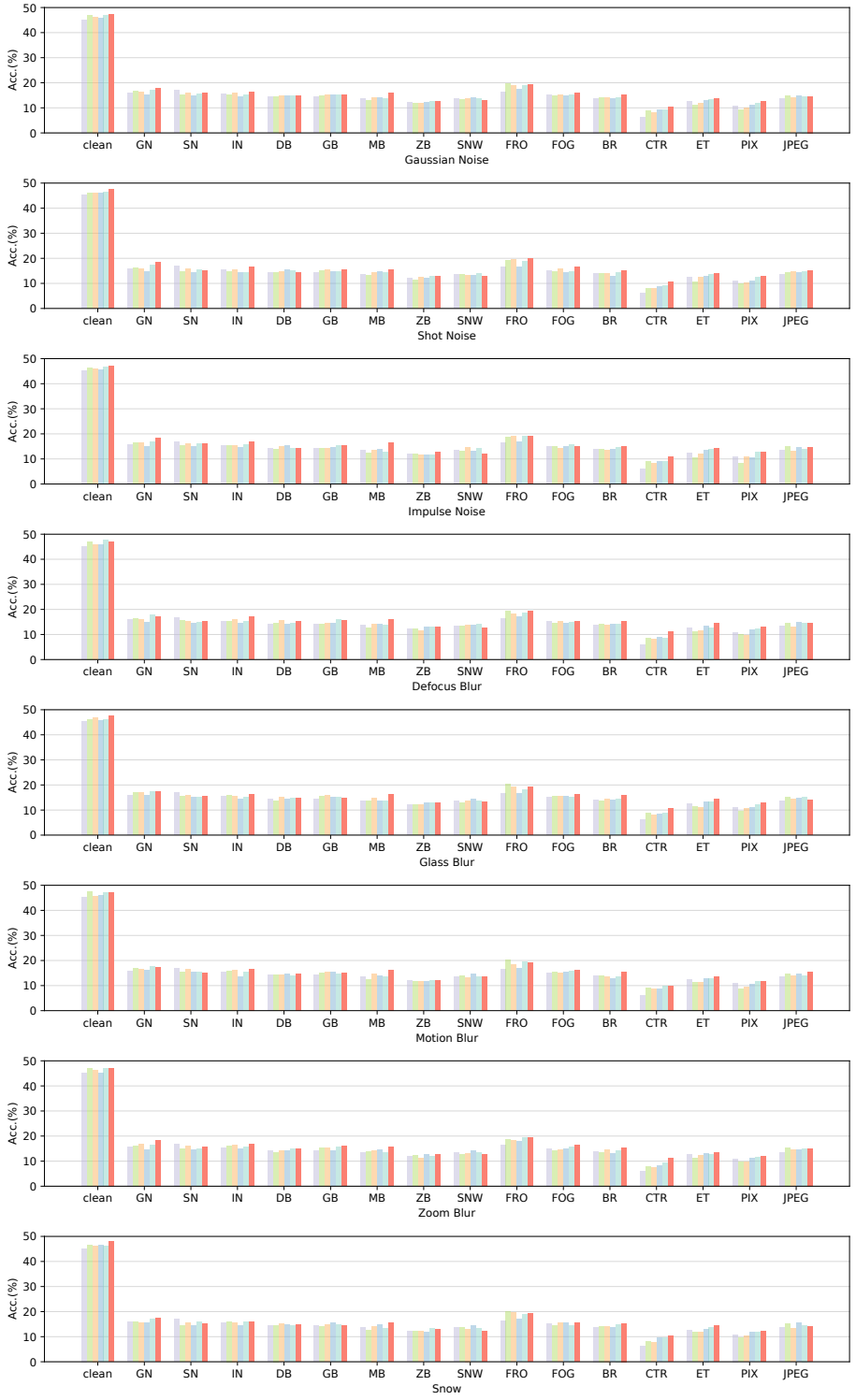


Fig. 6. Comparing the effectiveness and robustness of repairing methods on ResNet-18 by repairing the DNNs on one of the CIFAR-10's corruption dataset $\mathcal{D}_i^c$ (CIFAR-10-C) and evaluating on the other corruption dataset $\{\mathcal{D}_k^c | \mathcal{D}_k^c \in \mathcal{D}^c, k \neq i\}$.


Acc.(%)
0
10
20
30
40
50
clean
GN
SN
IN
DB
GB
MB
ZB
SNW
FRO
FOG
BR
CTR
ET
PIX
JPEG
Gaussian Noise
Shot Noise
Impulse Noise
Defocus Blur
Glass Blur
Motion Blur
Zoom Blur
Snow

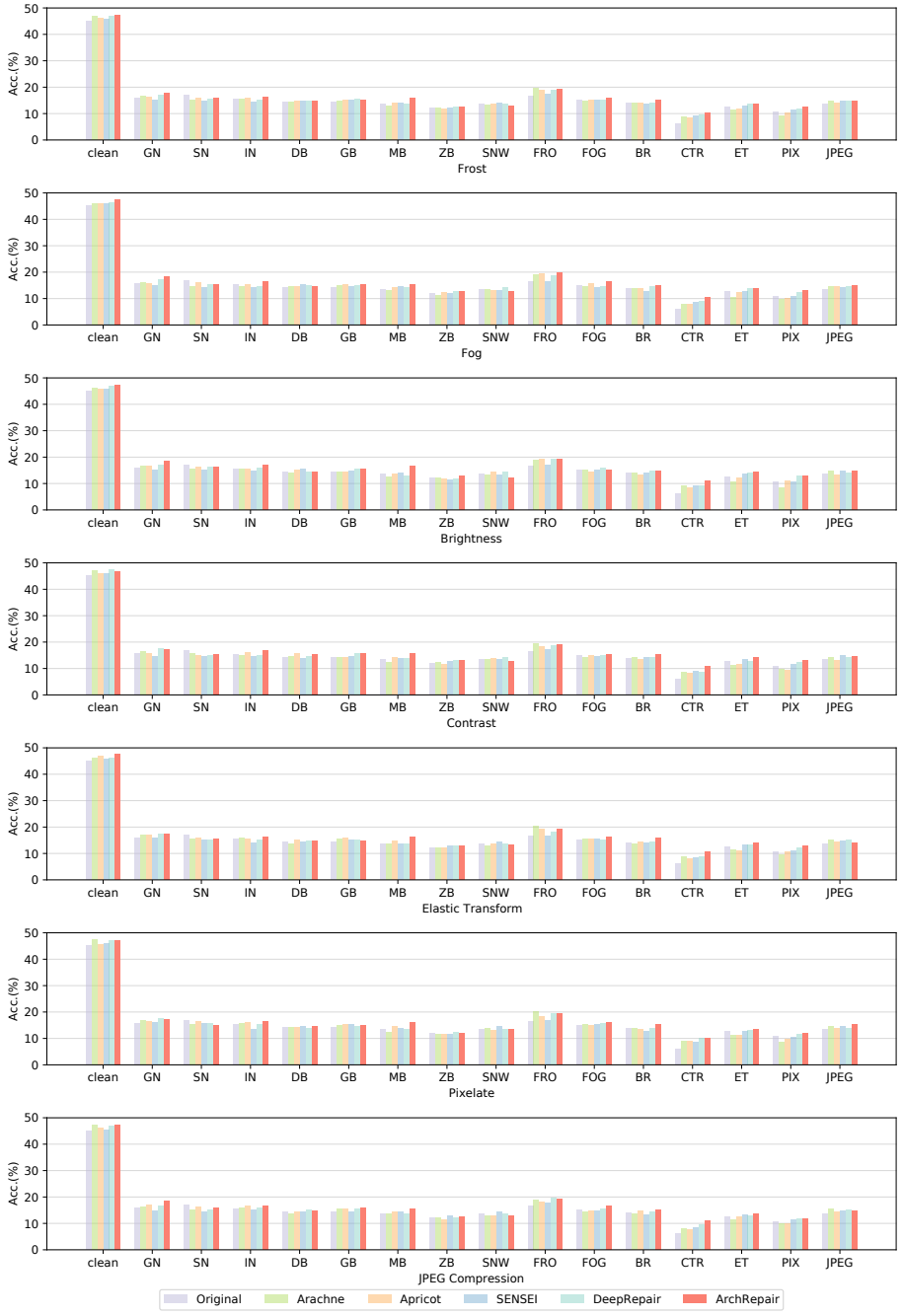


Fig. 7. Comparing the effectiveness and robustness of repairing methods on ResNet-18 by repairing the DNNs on one of the Tiny-Imagenet's corruption dataset $\mathcal{D}_i^c$ (Tiny-ImageNet-C) and evaluating on the other corruption dataset $\{\mathcal{D}_k^c | \mathcal{D}_k^c \in \mathcal{D}^c, k \neq i\}$.

Table 8. Average robust accuracy (%, repeated over 5 runs) of 6 different DNNs (*i.e.*, VGGNet-16, ResNet-18. ResNet-50, ResNet-101, DenseNet-121, and EfficientNet-B0) repaired on 4 corruption dataset (*i.e.*, CIFAR-10-C, Tiny-ImageNet-C, CIFAR-100-C, and ImageNet-C) by different repairing methods.

Baseline	CIFAR-10						Tiny-ImageNet					
	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0
Original	64.83	70.70	72.59	72.62	74.18	76.11	16.85	17.59	18.77	18.97	19.76	20.37
MODE [43]	64.25	71.22	72.63	73.99	75.09	76.66	16.97	17.82	18.77	19.35	19.76	20.79
Apricot [70]	64.77	71.75	72.95	73.17	74.76	75.83	17.13	17.63	18.95	19.35	20.65	20.84
Arachne [55]	65.72	71.63	72.11	72.76	74.28	75.27	17.23	18.25	19.19	19.37	21.21	21.77
SENSEI [18]	66.75	72.82	73.73	73.97	75.91	76.95	17.62	18.27	19.43	20.86	21.85	22.85
Few-Shot [50]	65.57	72.54	73.15	74.25	75.22	76.83	16.93	17.99	19.53	20.53	21.67	21.97
DeepRepair [68]	67.97	73.36	74.19	75.15	76.56	77.33	18.25	19.26	20.75	21.62	22.35	23.46
ArchRepair (ours)	67.49	74.25	74.57	76.03	76.56	78.56	18.77	19.53	20.62	21.88	22.97	23.66

Baseline	CIFAR-100						ImageNet					
	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0	VGG-16	Res-18	Res-50	Res-101	Den-121	Eff-B0
Original	32.59	33.72	34.59	35.66	36.75	37.82	27.55	28.66	29.35	30.53	31.13	33.69
MODE [43]	33.13	33.98	35.27	36.28	37.28	38.13	28.16	29.14	29.64	30.99	31.85	34.17
Apricot [70]	33.43	33.64	35.15	35.97	37.28	38.28	27.93	29.07	29.67	30.74	31.74	34.63
Arachne [55]	32.88	34.11	35.63	35.83	37.96	38.76	28.42	28.96	30.12	30.68	31.37	33.79
SENSEI [18]	33.85	34.89	36.72	36.90	38.54	40.22	28.67	29.11	30.64	31.75	32.44	34.88
Few-Shot [50]	33.67	34.67	36.72	36.83	37.44	38.25	28.45	29.32	30.59	31.33	32.75	35.09
DeepRepair [68]	34.14	35.82	37.67	37.96	39.25	40.33	29.36	29.89	31.86	32.86	33.16	35.85
ArchRepair (ours)	34.57	35.77	38.59	39.13	39.64	40.65	29.55	29.77	31.97	32.87	34.22	36.24

Table 9. Block suspiciousness $\mathcal{S}_{\mathrm{B}}$ under 8 different thresholds ϵ_i and the accuracy of 2 DNNs (*i.e.*, ResNet-18 and ResNet-50) repaired on 4 different blocks. Obviously repairing on the block with the highest block suspiciousness has the best performance.

	CIFAR-10									Tiny-ImageNet								
	Acc. (%) on $\mathcal{D}^{\mathrm{v}}$	Block Suspiciousness $\mathcal{S}_{\mathrm{B}}$								Acc. (%) on $\mathcal{D}^{\mathrm{v}}$	Block Suspiciousness $\mathcal{S}_{\mathrm{B}}$							
		ϵ_{10}	ϵ_{20}	ϵ_{30}	ϵ_{40}	ϵ_{50}	ϵ_{75}	ϵ_{100}	ϵ_{150}		ϵ_{10}	ϵ_{20}	ϵ_{30}	ϵ_{40}	ϵ_{50}	ϵ_{75}	ϵ_{100}	ϵ_{150}
Block 1	85.374	0	3	6	8	8	18	22	40	46.11	1	1	4	4	4	12	23	41
Block 2	86.377	0	0	0	1	1	2	5	16	46.29	0	1	2	2	2	6	9	16
Block 3	85.090	0	1	3	9	17	19	26	47	47.13	0	0	0	1	4	5	9	16
Block 4	88.294	10	20	21	22	24	48	48	50	47.35	9	18	24	33	40	52	60	79

(a) Block suspiciousness and repairing accuracy on ResNet-18

	CIFAR-10									Tiny-ImageNet								
	Acc. (%) on $\mathcal{D}^{\mathrm{v}}$	Block Suspiciousness $\mathcal{S}_{\mathrm{B}}$								Acc. (%) on $\mathcal{D}^{\mathrm{v}}$	Block Suspiciousness $\mathcal{S}_{\mathrm{B}}$							
		ϵ_{10}	ϵ_{20}	ϵ_{30}	ϵ_{40}	ϵ_{50}	ϵ_{75}	ϵ_{100}	ϵ_{150}		ϵ_{10}	ϵ_{20}	ϵ_{30}	ϵ_{40}	ϵ_{50}	ϵ_{75}	ϵ_{100}	ϵ_{150}
Block 1	82.115	1	2	2	4	4	7	7	7	45.83	0	0	0	0	0	0	0	0
Block 2	84.313	1	1	6	8	8	10	10	15	46.55	0	0	0	0	0	0	0	3
Block 3	89.576	8	18	24	32	42	58	86	139	47.82	10	20	30	40	48	67	84	119
Block 4	87.254	0	0	0	0	0	0	0	0	46.27	0	0	0	0	0	1	2	3

(b) Block suspiciousness and repairing accuracy on ResNet-50

It's worth mentioning that for a simpler DNN architecture, *i.e.*, ResNet-18, the vulnerable candidate block can be located more accurately when the threshold ϵ_i is small. As the threshold ϵ_i increases, the block suspiciousness $\mathcal{S}_{\mathrm{B}}$ on other blocks becomes larger, making the localization method difficult to identify the vulnerable block. While for ResNet-50 (a relatively complex DNN), no matter what value the threshold ϵ_i is, the localization result is always significantly accurate (with a much higher suspiciousness $\mathcal{S}_{\mathrm{B}}$ compared with other blocks).

Table 10. Comparing the two variants of our methods on four DNNs by evaluating the accuracy of repaired DNN under testing dataset $\mathcal{D}^t$.

	CIFAR-10				Tiny-ImageNet			
	ResNet-18	**ResNet-50**	**ResNet-101**	**DenseNet-121**	**ResNet-18**	**ResNet-50**	**ResNet-101**	**DenseNet-121**
Original	85.00	85.17	85.72	87.97	45.15	46.27	46.14	48.73
Layer-lv	85.02	85.26	85.29	89.86	45.35	45.11	45.84	46.17
Block-lv	88.29	89.58	90.38	91.37	47.35	47.82	46.73	46.84

Answer to RQ3: *ArchRepair* is able to locate the most vulnerable block regardless of the settings of threshold ϵ_i on different DNNs' architectures we evaluated (*e.g.*, ResNet-18 and ResNet-50).

5.4 RQ4: How different components of *ArchRepair* impact its overall performance?

To demonstrate the effectiveness of our *ArchRepair* and investigate how each component contributes to its overall performance, we conduct an ablation study by repairing 4 pre-trained models (*i.e.*, ResNet-18, ResNet-50, ResNet-101, and DenseNet-121) with two variants of our method on both CIFAR-10 and Tiny-ImageNet datasets. Table 10 summarizes the evaluation results. The first one performs *ArchRepair* on one single layer of the DNN, and we denote these variants as 'Layer-lv' in Table 10. The second one is our full (complete) version that applies *ArchRepair* at the block level, we denote this variant as 'Block-lv' in Table 10.

Compared with the original DNNs, the performance of 'Layer-lv' is acceptable on CIFAR-10 dataset, as it slightly improves the behaviors on three DNNs (*i.e.*, ResNet-18, ResNet-50, and DenseNet-121) and only decreases slightly on ResNet-101. The 'Block-lv' achieves better performance on all of the four DNNs on CIFAR-10, and these results indicate that *ArchRepair*'s repairing capability is effective at both levels. The performance on 'Block-lv' is better than the 'Layer-lv' on all the four DNNs on two different datasets, especially on the more challenging dataset Tiny-ImageNet, where 'Layer-lv' only shows small improvement on ResNet-18 while 'Block-lv' has significant improvement on all three variants of ResNet. This demonstrates that repairing on one specific layer cannot fully unleash *ArchRepair*'s potential while repairing on a block enables to take the advantage of all components of *ArchRepair*. Note that even though both 'Block-lv' and 'Layer-lv' fail to repair DenseNet-121 on Tiny-ImageNet (as well as all the SOTA baseline methods, see evaluation results in Table 5), 'Block-lv' still performs better than 'Layer-lv'.

Answer to RQ4: Block-level repairing is more effective than layer-level one towards fully releasing *ArchRepair*'s repairing capability. In addition, adjusting the network's architecture and weights simultaneously is more effective than only adjusting the weights, especially for block-level repairing, demonstrating that jointly repairing the block architecture and weights is a promising research direction for DNN repair.

5.5 Threat to validity

The threats to the validity of this paper could come from the following aspects: 1) The selected dataset and the used model architectures could be a threat. To mitigate this, we selected popular datasets as well as diverse architectures to evaluate our method. 2) The selection of the corruption dataset could be biased, *i.e.*, our method and results may not generalize well on other corruptions. To counteract this, we tried our best and selected as many as 15 diverse and commonly used natural corruptions in the standard benchmarks of previous work [25]. 3) Another threat is from

the implementation of our method as well as the usage of the existing baselines. To mitigate the threat, we carefully follow the configuration as stated in the original papers or implementation, respectively. Moreover, our co-authors carefully test and review our code and the configuration of other tools. Furthermore, to be comprehensive for better understanding the position of *ArchRepair*, we perform a large-scale comparative study against 6 SOTA DNN repair techniques. The results confirm DNN repair could be even more promising and there are still opportunities ahead when going beyond focusing on repairing DNN weights only.

6 RELATED WORK

6.1 DNN Testing

DNN testing is an important and relevant technique to DNN repair, aiming to detect potential buggy issues of a DNN. Some recent work focuses on testing criteria design. For example, DeepXplore [46] proposes the neuron coverage based on the number of activated neurons on given testing data, where the neuron coverage represents the adequacy of the testing data. Similarly, DeepGauge [41] proposes multi-granularity testing criteria, which are based on the analysis of neural behaviors. DeepCT [40] considers the interactions between the different neurons, and further Kim *et al.* [30] propose the coverage criteria to measure the surprise of the inputs based on the neuron features at the layer level. Some researchers [23, 52] recently also point out that the neuron coverage might fail if most of the neurons are activated by a few test cases, and further in-depth research is still needed along this line.

Overall, these testing criteria lay the early foundation for testing generation techniques to detect defects in DNNs. DeepTest [61] generates test cases based on the guidance of neuron coverage. TensorFuzz [45] proposes a distance-based coverage-guided fuzzing techniques to test DNNs. Similarly, DeepHunter [65] proposes another coverage-guided testing technique by integrating the coverage criteria from DeepGauge. Readers can also see [42]. DeepStellar [12] employs the coverage criteria and fuzzing technique, to test and analyze the recurrent neural network. More discussions on the progress of machine learning testing can be referred to the recent survey [39, 71]. Different from these testing techniques, our work mainly focuses on repairing DNNs and enhancing their robustness and generalization capability, which can be considered as the downstream tasks of DNN testing.

6.2 Fault Localization on Deep Neuron Network

Fault localization aims to locate the root causes of software failures. Similar approaches have been widely studied for traditional software, which focuses on developing faults identification methods such as spectral-based [1, 28, 32, 33, 44, 47, 72], model-based [4, 51], slice-based [2], and semantic fault localization [9]. Several works recently introduce fault localization on DNNs to find vulnerable neurons and repair their weights. Representative techniques include sensitivity-based fault localization [55] and spectrum-based fault localization [13]. Eniser *et al.* [13] try to identify suspicious neurons responsible for unsatisfactory DNN performance, which is an early attempt to introduce fault localization techniques on DNNs with promising results. However, these methods only consider a fixed DNN architecture and neuron-aware buggy behaviors, which is less flexible for real-world applications. Our work repairs DNN at a higher level (*i.e.*, block level) by localizing the vulnerable block and jointly repairing the block architecture and weights, which is novel and has not been investigated in previous work.

6.3 DNN Repair

So far, there are several attempts for repairing DNN models. Inspired by software debugging, Ma *et al.* [43] propose a novel model debugging technique for neural network models, which is denoted as MODE. MODE first performs state differential analysis on hidden layers to identify the faulty neurons that are responsible for the misclassification. Then, an input selection algorithm is used to select new input samples to retrain the faulty neurons.

Zhang *et al.* [70] propose a weight-adjustment approach named Apricot to fix the DNN. Apricot first generates a set of reduced DNNs from the original model and trains them with a random subset of the original training dataset, respectively. For each failure example, Apricot separates reduced DNN models into two partitions, one successfully predicts the label and the other does not, and takes the mean of the corresponding weight assignments of two partitions. After that, Apricot automatically adjusts the weight with these mean values. Further, Sohn *et al.* [55] propose a search-based repair technique for DNNs, called Arachne. Unlike other techniques, Arachne directly manipulates the neuron weights without retraining. Arachne first uses positive and negative input data to retain correct behavior and generate a patch, respectively. Then uses Particle Swarm Optimization (PSO) to search and locate faulty neurons, and uses the result of PSO candidate to update neurons' weights, and further calculates fitness value based on the outcomes.

Recently, Gao *et al.* [18] have proposed a new algorithm called SENSEI, which uses guided test generation techniques to address the data augmentation problem for robust generalization of DNNs under natural environmental variations. Firstly, SENSEI uses a genetic search on a space of the natural environmental variants of each training input data to identify the worst variant for augmentation on each epoch. Besides, SENSEI uses a heuristic technique named selective augmentation, which allows skipping augmentation in certain epochs based on an analysis of the DNN's current robustness. Another recent attempt for DNN repair is DeepRepair [68], a method that repairs the DNN on the image classification task. DeepRepair uses a style-guided data augmentation for DNN repairing to introduce the unknown failure patterns into the training data to retrain the model and applies clustering-based failure data generation to improve the effectiveness of data augmentation.

Our repairing method is orthogonal to existing data-augmentation based methods such as SENSEI [18] and DeepRepair [68], where we focus on repairing DNN from the architecture and weight perspective. Our method also goes one step further beyond the weight level (*e.g.*, MODE [43], Apricot [70], and Arachne [55]), and considers at a higher granularity by jointly repairing architecture and weights at the block level, which is demonstrated to be a promising direction for DNN repairing.

Note that, the field of DNN repairing has been progressing very fast, with some concurrent work proposed during the enhancement of our work. We would continuously update our supplementary website [56] to keep the relevant techniques of DNN repairing updated, and hopefully provide a basis to ease further research in this direction.

6.4 Neural Architecture Search

Neural architecture search (NAS) could be another relevant line of our work, aiming to automatically design an architecture instead of handcrafting one. Typical NAS includes evolution-based [49, 64], and reinforcement-learning-based [3] methods. However, the resources RL or evolution-based methods leveraged are often very expensive and still unaffordable in practice. More recently, DARTS [37] relaxes the search space to make it continuous so that the search processes can be performed based on the gradient. Differentiable NAS approaches can significantly reduce

computational costs. Our search method is based on PC-DARTS [66], a stability-improved variant of DARTS by introducing a partially connected mechanism.

The purpose of repairing and NAS is very different. The former intends to fix the buggy behaviors that follow some patterns with generalization capability, while NAS is to design general architecture automatically for better performance (*e.g.*, energy efficiency). In this paper, we formulate the block-level joint architecture and weight repairing as a NAS problem, which demonstrates the possibilities and chances for DNN repair along this direction.

7 CONCLUSION

In this work, we have proposed *ArchRepair*, an architecture-oriented DNN repair at block level, which offers a good trade-off between repaired network accuracy and time consumption, compared to neuron-level, layer-level, and network-level (data augmentation) repairing. To achieve this, two key problems are identified and solved sequentially, *i.e.*, *block localization*, and *joint architecture and weights repairing*. By jointly repairing both architecture and weights on the candidate block for repairing, ***ArchRepair*** is able to achieve competitive performance compared with 6 SOTA techniques. Our extensive evaluation has also demonstrated that *ArchRepair* could not only enhance the accuracy but also the robustness across various corruption patterns while being cost-effective. To the best of our knowledge, this work is among the very early attempt at DNN repair by considering adjusting both the architecture and weights at the 'block-level'. Our research also initiates a promising direction for further DNN repair research, towards addressing the current urgent industrial demands for reliable and trustworthy DNN deployment in diverse real-world environments.

ACKNOWLEDGMENT

This work was supported in part by JST-Mirai Program Grant No.JPMJMI20B8, JSPS KAKENHI Grant No.JP20H04168, No.JP21H04877, JST the establishment of university fellowships towards the creation of science technology innovation, Grant No.JPMJFS2132, as well as Canada CIFAR AI Chairs Program and the Natural Sciences and Engineering Research Council of Canada (NSERC No.RGPIN-2021-02549, No.RGPAS-2021-00034, No.DGECR-2021-00019).

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (Nov. 2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035

[2] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. 2011. Fault-localization using dynamic slicing and change impact analysis. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. 520–523. https://doi.org/10.1109/ASE.2011.6100114 ISSN: 1938-4300.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing Neural Network Architectures using Reinforcement Learning. *arXiv:1611.02167 [cs]* (March 2017). arXiv: 1611.02167.

[4] Geoff Birch, Bernd Fischer, and Michael Poppleton. 2019. Fast test suite-driven model-based fault localisation with application to pinpointing defects in student programs. *Software & Systems Modeling* 18, 1 (Feb. 2019), 445–471. https://doi.org/10.1007/s10270-017-0612-y

[5] Yun Chen, Frieda Rong, Shivam Duggal, Shenlong Wang, Xinchen Yan, Sivabalan Manivasagam, Shangjie Xue, Ersin Yumer, and Raquel Urtasun. 2021. GeoSim: Realistic Video Simulation via Geometry-Aware Composition for Self-Driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7230–7240.

[6] Yupeng Cheng, Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Shang-Wei Lin, Weisi Lin, Wei Feng, and Yang Liu. 2021. Pasadena: Perceptually Aware and Stealthy Adversarial Denoise Attack. *IEEE Transactions on Multimedia (TMM)* (2021).

[7] Yupeng Cheng, Felix Juefei-Xu, Qing Guo, Huazhu Fu, Xiaofei Xie, Shang-Wei Lin, Weisi Lin, and Yang Liu. 2020. Adversarial exposure attack on diabetic retinopathy imagery. *arXiv preprint arXiv:2009.09231* (2020).

[8] Chiho Choi, Joon Hee Choi, Jiachen Li, and Srikanth Malla. 2021. Shared Cross-Modal Trajectory Prediction for Autonomous Driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

244–253.

[9] Maria Christakis, Matthias Heizmann, Muhammad Numair Mansur, Christian Schilling, and Valentin Wüstholz. 2019. Semantic Fault Localization and Suspiciousness Ranking. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 226–243. https://doi.org/10.1007/978-3-030-17462-0_13

[10] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[11] Brian Dolhansky, Joanna Bitton, Ben Pflaum, Jikuo Lu, Russ Howes, Menglin Wang, and Cristian Canton Ferrer. 2020. The deepfake detection challenge dataset. *arXiv e-prints* (2020), arXiv–2006.

[12] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: Model-Based Quantitative Analysis of Stateful Deep Learning Systems. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 477–487. https://doi.org/10.1145/3338906.3338954

[13] Hasan Ferit Eniser, Simos Gerasimou, and Alper Sen. 2019. DeepFault: Fault Localization for Deep Neural Networks. In *Fundamental Approaches to Software Engineering*, Reiner Hähnle and Wil van der Aalst (Eds.). Springer International Publishing, Cham, v.

[14] Deng-Ping Fan, Tao Zhou, Ge-Peng Ji, Yi Zhou, Geng Chen, Huazhu Fu, Jianbing Shen, and Ling Shao. 2020. Inf-net: Automatic covid-19 lung infection segmentation from ct images. *IEEE Transactions on Medical Imaging* 39, 8 (2020), 2626–2637.

[15] Ruijun Gao, Qing Guo, Felix Juefei-Xu, Hongkai Yu, and Wei Feng. 2021. AdvHaze: Adversarial Haze Attack. *arXiv preprint arXiv:2104.13673* (2021).

[16] Ruijun Gao, Qing Guo, Felix Juefei-Xu, Hongkai Yu, Xuhong Ren, Wei Feng, and Song Wang. 2020. Making images undiscoverable from co-saliency detection. *arXiv preprint arXiv:2009.09258* (2020).

[17] Ruijun Gao, Qing Guo, Qian Zhang, Felix Juefei-Xu, Hongkai Yu, and Wei Feng. 2021. Adversarial Relighting against Face Recognition. *arXiv preprint arXiv:2108.07920* (2021).

[18] Xiang Gao, Ripon K. Saha, Mukul R. Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing Based Data Augmentation to Improve Robustness of Deep Neural Networks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1147–1158. https://doi.org/10.1145/3377811.3380415

[19] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic software repair: A survey. *IEEE Transactions on Software Engineering* 45, 1 (2017), 34–67.

[20] Qing Guo, Ziyi Cheng, Felix Juefei-Xu, Lei Ma, Xiaofei Xie, Yang Liu, and Jianjun Zhao. 2021. Learning to Adversarially Blur Visual Object Tracking. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE.

[21] Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, Jian Wang, Bing Yu, Wei Feng, and Yang Liu. 2020. Watch out! Motion is Blurring the Vision of Your Deep Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[22] Qing Guo, Xiaofei Xie, Felix Juefei-Xu, Lei Ma, Zhongguo Li, Wanli Xue, Wei Feng, and Yang Liu. 2020. SPARK: Spatial-aware Online Incremental Attack Against Visual Tracking. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[23] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. 2020. Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks?. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[24] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[25] Dan Hendrycks and Thomas Dietterich. 2019. Benchmarking Neural Network Robustness to Common Corruptions and Perturbations. *Proceedings of the International Conference on Learning Representations* (2019).

[26] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[27] Yihao Huang, Felix Juefei-Xu, Qing Guo, Weikai Miao, Yang Liu, and Geguang Pu. 2021. AdvBokeh: Learning to Adversarially Defocus Blur. *arXiv preprint* (2021).

[28] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[29] Felix Juefei-Xu, Run Wang, Yihao Huang, Qing Guo, Lei Ma, and Yang Liu. 2021. Countering malicious deepfakes: Survey, battleground, and horizon. *arXiv preprint arXiv:2103.00218* (2021).

[30] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

[31] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. *University of Toronto* (2009), 32–33.

[32] David Landsberg, Hana Chockler, Daniel Kroening, and Matt Lewis. 2015. Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Alexander Egyed and Ina Schaefer (Eds.). Springer, Berlin, Heidelberg, 115–129. https://doi.org/10.1007/978-3-662-46675-9_8

[33] David Landsberg, Youcheng Sun, and Daniel Kroening. 2018. Optimising Spectrum Based Fault Localisation for Single Fault Programs Using Specifications. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Alessandra Russo and Andy Schürr (Eds.). Springer International Publishing, Cham, 246–263. https://doi.org/10.1007/978-3-319-89363-1_14

[34] Ya Le and X. Yang. 2015. Tiny ImageNet Visual Recognition Challenge. In *Stanford CS 231N*.

[35] Yiming Li, Congcong Wen, Felix Juefei-Xu, and Chen Feng. 2021. Fooling LiDAR Perception via Adversarial Trajectory Perturbation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[36] Yiming Li, Congcong Wen, Felix Juefei-Xu, and Chen Feng. 2021. Fooling LiDAR Perception via Adversarial Trajectory Perturbation. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. IEEE.

[37] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. *arXiv preprint arXiv:1806.09055* (2018).

[38] Chenxu Luo, Xiaodong Yang, and Alan Yuille. 2021. Self-Supervised Pillar Motion Learning for Autonomous Driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 3183–3192.

[39] Lei Ma, Felix Juefei-Xu, Minhui Xue, Qiang Hu, Sen Chen, Bo Li, Yang Liu, Jianjun Zhao, Jianxiong Yin, and Simon See. 2018. Secure Deep Learning Engineering: A Software Quality Assurance Perspective. *arXiv preprint arXiv:1810.04538* (2018).

[40] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 614–618.

[41] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[42] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.

[43] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: automated neural network model debugging via state differential analysis and input selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 175–186.

[44] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3 (Aug. 2011), 11:1–11:32. https://doi.org/10.1145/2000791.2000795

[45] Augustus Odena and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the Thirty-sixth International Conference on Machine Learning*.

[46] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[47] Alexandre Perez, Rui Abreu, and Arie van Deursen. 2017. A Test-Suite Diagnosability Metric for Spectrum-Based Fault Localization Approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 654–664. https://doi.org/10.1109/ICSE.2017.66 ISSN: 1558-1225.

[48] Aditya Prakash, Kashyap Chitta, and Andreas Geiger. 2021. Multi-Modal Fusion Transformer for End-to-End Autonomous Driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 7077–7087.

[49] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. *arXiv:1802.01548 [cs]* (Feb. 2019). arXiv: 1802.01548.

[50] Xuhong Ren, Bing Yu, Hua Qi, Felix Juefei-Xu, Zhuo Li, Wanli Xue, Lei Ma, and Jianjun Zhao. 2020. Few-Shot Guided Mix for DNN Repairing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 717–721. https://doi.org/10.1109/ICSME46990.2020.00079

[51] Erickson H. da S. Alves, Lucas C. Cordeiro, and Eddie B. de L. Filho. 2017. A method to localize faults in concurrent C programs. *Journal of Systems and Software* 132 (Oct. 2017), 336–352. https://doi.org/10.1016/j.jss.2017.03.010

[52] Jasmine Sekhon and Cody Fleming. 2019. Towards improved testing for deep learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 85–88.

[53] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015*, Yoshua Bengio and Yann LeCun (Eds.).

[54] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[55] Jeongju Sohn, Sungmin Kang, and Shin Yoo. 2019. Search Based Repair of Deep Neural Networks. *arXiv preprint arXiv:1912.12463* (2019).

[56] The supplementary website of ArchRepair paper. 2023. https://sites.google.com/view/archrepair.

[57] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. (2019). https://doi.org/10.48550/ARXIV.1905.11946

[58] Binyu Tian, Qing Guo, Felix Juefei-Xu, Wen Le Chan, Yupeng Cheng, Xiaohong Li, Xiaofei Xie, and Shengchao Qin. 2021. Bias Field Poses a Threat to DNN-Based X-Ray Recognition. In *IEEE International Conference on Multimedia and Expo (ICME)*.

[59] Binyu Tian, Felix Juefei-Xu, Qing Guo, Xiaofei Xie, Xiaohong Li, and Yang Liu. 2021. AVA: Adversarial Vignetting Attack against Visual Recognition. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.

[60] Yuchi Tian. 2020. Repairing Confusion and Bias Errors for DNN-Based Image Classifiers. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1699–1700.

[61] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[62] Jingkang Wang, Ava Pun, James Tu, Sivabalan Manivasagam, Abbas Sadat, Sergio Casas, Mengye Ren, and Raquel Urtasun. 2021. AdvSim: Generating Safety-Critical Scenarios for Self-Driving Vehicles. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 9909–9918.

[63] Run Wang, Felix Juefei-Xu, Qing Guo, Yihao Huang, Xiaofei Xie, Lei Ma, and Yang Liu. 2020. Amora: Black-box Adversarial Morphing Attack. In *Proceedings of the ACM International Conference on Multimedia (ACM MM)*.

[64] Lingxi Xie and Alan Yuille. 2017. Genetic CNN. *arXiv:1703.01513 [cs]* (March 2017). arXiv: 1703.01513.

[65] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. 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*. 146–157.

[66] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. 2020. PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search. *arXiv:1907.05737 [cs]* (April 2020). arXiv: 1907.05737.

[67] Jason Yim, Reena Chopra, Terry Spitz, Jim Winkens, Annette Obika, Christopher Kelly, Harry Askham, Marko Lukic, Josef Huemer, Katrin Fasler, et al. 2020. Predicting conversion to wet age-related macular degeneration using deep learning. *Nature Medicine* 26, 6 (2020), 892–899.

[68] Bing Yu, Hua Qi, Qing Guo, Felix Juefei-Xu, Xiaofei Xie, Lei Ma, and Jianjun Zhao. 2021. DeepRepair: Style-Guided Repairing for Deep Neural Networks in the Real-World Operational Environment. *IEEE Transactions on Reliability* (2021), 1–16. https://doi.org/10.1109/TR.2021.3096332

[69] Liming Zhai, Felix Juefei-Xu, Qing Guo, Xiaofei Xie, Lei Ma, Wei Feng, Shengchao Qin, and Yang Liu. 2020. It's Raining Cats or Dogs? Adversarial Rain Attack on DNN Perception. *arXiv preprint arXiv:2009.09205* (2020).

[70] Hao Zhang and W.K. Chan. 2019. Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 376–387. https://doi.org/10.1109/ASE.2019.00043

[71] J. M. Zhang, M. Harman, L. Ma, and Y. Liu. 2020. Machine Learning Testing: Survey, Landscapes and Horizons. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2019.2962027

[72] Long Zhang, Lanfei Yan, Zhenyu Zhang, Jian Zhang, W. K. Chan, and Zheng Zheng. 2017. A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization. *Journal of Systems and Software* 129 (July 2017), 35–57. https://doi.org/10.1016/j.jss.2017.04.017