DeepPatch

Maintaining Deep Learning Model Programs to Retain Standard Accuracy with Substantial Robustness Improvement

WEI, Zhengyuan; WANG, Haipeng; ASHRAF, Imran; CHAN, Wing-Kwong

Published in:
ACM Transactions on Software Engineering and Methodology

Published: 01/09/2023

Document Version:
Post-print, also known as Accepted Author Manuscript, Peer-reviewed or Author Final version

Published version (DOI):
10.1145/3604609

Publication details:
WEI, Z., WANG, H., ASHRAF, I., & CHAN, W-K. (2023). DeepPatch: Maintaining Deep Learning Model Programs to Retain Standard Accuracy with Substantial Robustness Improvement. *ACM Transactions on Software Engineering and Methodology*, *32*(6), Article 150. Advance online publication. https://doi.org/10.1145/3604609

DeepPatch: Maintaining Deep Learning Model Programs to Retain Standard Accuracy with Substantial Robustness Improvement *†

Zhengyuan Wei, Haipeng Wang, Imran Ashraf, and W.K. Chan ‡

Abstract

Maintaining a deep learning (DL) model by making the model substantially more robust through retraining with plenty of adversarial examples of non-trivial perturbation strength often reduces the model's standard accuracy. Many existing model repair or maintenance techniques sacrifice standard accuracy to produce a large gain in robustness or vice versa. This paper proposes DeepPatch, a novel technique to maintain filter-intensive DL models. To the best of our knowledge, DeepPatch is the first work to address the challenge of standard accuracy retention while substantially improving the robustness of DL models with plenty of adversarial examples of non-trivial and diverse perturbation strengths. Rather than following the conventional wisdom to generalize all the components of a DL model over the union set of clean and adversarial samples, DeepPatch formulates a novel division of labor method to adaptively activate a subset of its inserted processing units to process individual samples. Its produced model can generate the original or replacement feature maps in each forward pass of the patched model, making the patched model carry an intrinsic property of behaving like the model under maintenance on demand. The overall experimental results show that DeepPatch successfully retains the standard accuracy of all pretrained models while improving the robustness accuracy substantially. On the other hand, the models produced by the peer techniques suffer from either large standard accuracy loss or small robustness improvement compared with the models under maintenance, rendering them unsuitable in general to replace the latter.
Keywords: model testing, maintenance, accuracy recovery

1 Introduction

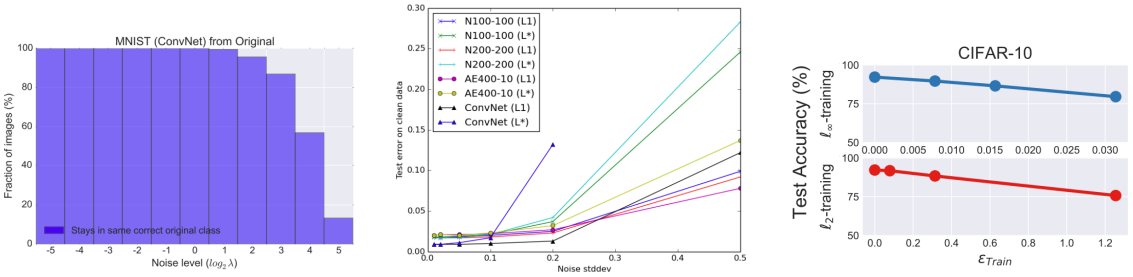
Applications [1–3] with deep learning (DL) models as components are widely used in practice with impressive performance. Examples include autonomous driving cars [1], image recognition [2], and medical image diagnosis [3]. In some applications, their performances outperform human experts [4, 5]. At the same time, such models incur various problems. For instance, DL models can be easily fooled by adversarial examples [6], which contain perturbations from the clean samples, or behave biasedly with respect to certain data fairness criteria [7]. Such discoveries motivate researchers to study how to evaluate, assure, and improve the correctness of pretrained DL models. For instance, challenges in software engineering problems, such as test oracles [8, 9], fuzzing [10], system testing [11, 12], mutation testing [13], repair [14–19], test adequacy [20–22], test case generation [23], selection [24], prioritization [25–27], robustness improvement [28], and model finetuning [29], are being studied.

DL models are well-known to be vulnerable to adversarial samples [6]. Improving the robustness of DL models is attractive, and a recent survey shows that it remains an open problem [30]. For instance, wrong predictions on natural robustness scenarios may lead to severe consequences such as causality [31]. A vehicle safety report [32] summarizes that there is one accident for about every two to four million miles. Moreover, for the image classification task, the one-pixel attack [33] represents the least perturbation on the sample to fool the DL models successfully. The general and classical FGSM [34] and PGD [35] attack methods can easily generate adversarial samples with human-imperceptible perturbations. Engstrom et. al [36] show that they can turn correctly predicted samples into adversarial examples by simply applying rotations and translations alone. Moreover, Gu et. al [37] find that the robustness of many DL models

*Manuscript accepted by *ACM Transactions on Software Engineering and Methodology (TOSEM)*. DOI: `https://doi.org/10.1145/3604609`

†This research is partly supported by the CityU MF˙EXT (project no. 9678180).

‡Z. Wei, H. Wang, I. Ashraf, and W.K. Chan were with the Department of Computer Science, City University of Hong Kong, Hong Kong, China.



(a) The prediction remains in the correct class (i.e., high accuracy) when the noise level is small (e.g., $\log_2 \lambda <= 0$). The height of the bar decreases quickly when the noise level becomes larger. The plot is originally Fig. 3(d) in [47].

(b) Test error on clean data increases rapidly when the noise standard deviation is larger than a small threshold (e.g., 0.2). The plot is originally the left subfigure of Fig. 1 in [37]

(c) Test accuracy on clean data drops gradually along with the increased strength of the adversary. The plot is originally Fig. 7 in [46].

Figure 1: Effects of adversarial training in literature.

decreases as the perturbation strength increases. Still, the generalization ability of a DL model to predict correctly on an arbitrary sample similar to the training dataset is the core reason for using the DL model in practice.

Generally speaking, adversarial training trains a DL model over an adversarial dataset [38]. However, many studies report that it generally lowers the model's performance in inferring clean samples [39, 40]. For instance, with the target of improving the robustness to 45% on ResNet18 [41] pretrained on the CIFAR10 dataset [42], previous studies [43] show that the loss in standard accuracy, which measures the proportion of samples in a clean dataset predicted by a model same as the ground truth labels of the respective samples, can be as high as 15%, which is severe. Aiming at an even higher degree of robustness while retaining the standard accuracy is important but difficult. In our experiment, none of the peer techniques can retain the standard accuracy while attaining robustness accuracy higher than 45% on the CIFAR10 dataset. In fact, they result in losses in standard accuracy of more than 15%. Rather than able to retain the standard accuracy, a severe tradeoff between the standard accuracy and the robustness[1] is a well-known problem [44] and widely observed across different techniques in their experiments [39, 45, 46].

Fig. 1 recaps the empirical results reported in the literature about both standard and adversarial training for the image classification task, which illustrates the difficulties in retaining standard accuracy while processing noisy samples. Fig. 1a shows that standard training originally can defend against a certain level of perturbation strength. However, the defense effect drops dramatically when the perturbation strength increases. Fig. 1b shows that by using adversarial examples with larger perturbation strengths in model training (larger x-value), the test errors on the clean samples of the DL models increase sharply (larger y-value) if the perturbation strength exceeds a small threshold (e.g., 0.2 in the plot). Fig. 1c further shows that adversarial training harms the standard accuracy to a severe level (e.g., from higher than 90% to around 75% in either measure) when the perturbation strength is large.

To keep the standard accuracy of the retrained models to a level close to the original DL model, we observe that in their experiments [19, 28, 48], many existing repair or maintenance techniques have to limit their adversarial datasets only to contain samples with a small perturbation strength. How to use a large set of adversarial examples with diverse (small and large) perturbation strengths for retraining a DL model under maintenance for substantial robustness improvement while retaining the standard accuracy of the DL model is still an open problem. To the best of our knowledge, this paper presents the first work toward solving this open problem, a problem often overlooked by existing techniques. To address this problem with an arbitrary model architecture is ambitious and represents an ultimate goal

[1]Standard accuracy and robustness accuracy are the percentages of samples in a test dataset and a robustness dataset predicted the same as the ground truth of respective samples. A typical example of a robustness dataset is a dataset containing adversarial examples, each produced from a sample in the test dataset with some perturbation.

for this line of research. As the first step, we target the image classification task, and the major DNN architecture used in this task is the convolutional neural network (CNN) models [49].

This paper presents DeepPatch, a novel technique to maintain filter-intensive DL models (convolutional neural network models) to address the above-mentioned problem. DeepPatch is formulated as a sequence of three tasks: filter assessment, model patching, and standard accuracy recovery. These tasks are assisted by our patching units, α-, β-, and γ-patching units (PUs for short), respectively. The first task, filter assessment, identifies a subset of filters with lower impacts on robustness. Then, the next task is model patching on the identified filters. A main design goal is to ensure DeepPatch has the ability to re-generate the original feature maps of the model under maintenance when processing with replacement filters. The last task is standard accuracy recovery. DeepPatch configures a metric computation unit with a separation boundary that aims to distinguish the samples more like the original or the perturbed validation datasets through its calibration process. It also organizes all the patching units in the constructing model in a master-slaves design style, where the γ-PU serves as the master and each β-PU serves as a slave.

In the inference time when using the patched model to process a sample, the master (γ-PU) is responsible for invoking the metric computation unit and decides a particular side to take, and broadcasts the decision to all β-PU in subsequent layers to select the corresponding subset of filters for execution to generate feature maps accordingly. As such, these patching units work like decision nodes with conditional branches in a patched model. The patched model can adaptively reproduce the full sets of the original feature maps or generate the replacement counterparts. The design made by DeepPatch on the patched model also implements a strict separation of the replacement filters from their original filter counterparts for model retraining and prediction. It makes the patched model output by DeepPatch has an intrinsic property of retaining the standard accuracy of the original model, which sets DeepPatch apart from existing techniques (that alter the full weights of the original models).

We have developed an end-to-end tool to implement DeepPatch. The DeepPatch tool accepts the original pretrained model instance (containing model data structure and model weights), an augmented training dataset instance, and both the original and augmented validation dataset instances as input. It generates the patched model instance at the end of the above-mentioned third task and saves it into files (including both model architecture and weight matrices), which can be loaded back into the program.

We have experimented DeepPatch on retraining datasets containing adversarial examples with a large range of perturbation strengths to train the DL models under maintenance. To align with the central problem addressed by this work, the evaluation also shows both aspects side by side: retention of standard accuracy and improvement in robustness accuracy.

The experimental result on four datasets, six pretrained models, and six techniques shows that DeepPatch retains the standard accuracy of the pretrained models with a significant boost in robustness. The peer techniques deteriorate the standard accuracy significantly or cannot attain significant robustness improvement, making them incur obvious limitations to replace the original pretrained models for use in general. With a substantial advantage in standard accuracy or robustness improvement over the peer techniques, DeepPatch outperforms them in the sum of standard accuracy and robustness accuracy. The experiment also shows that two pipelines of DeepPatch can generate effective results, showing the applicability of DeepPatch to integrate with other techniques for further advancing state of the art. We also report an exploratory study on the feasibility of applying DeepPatch on Vision Transformer (ViT) [50], a representative Transformer model for the image classification task. The result shows that the patched model produced by DeepPatch can retain the standard accuracy of ViT and improves the robustness accuracy to a large extent.

The main contribution of this paper is threefold:

1. This paper is the first work to show the feasibility of retaining the standard accuracy of the models under maintenance with significant robustness improvement on retraining datasets containing plenty of adversarial examples with large and diverse perturbation strengths.

2. It presents a novel technique DeepPatch. DeepPatch is the first work capable of reproducing the original feature maps of the model under maintenance after retraining to escape from the effects of training on adversarial examples.

3. The evaluation and the tool implementation show the feasibility and high effectiveness of DeepPatch.

The rest of the paper is organized as follows. Section 2 gives the preliminaries of DeepPatch. DeepPatch is presented in Section 3 followed by its evaluation in Section 4. Section 5 reviews closely related

work. We conclude this paper in Section 6.

2 Preliminaries

Table 1: The key symbols used in DeepPatch.

Name	Symbol	Meaning
Deep learning model	$\mathcal{F}$	A deep learning model under maintenance.
Convolutional layer	f_i	An i-th convolutional layer of $\mathcal{F}$.
Filters	G_i	A sequence of filters in f_i.
Filter	g_j	A filter in the convolutional layer f_i at the $j-$th position in G_i.
Blamed filters	$G_\ominus$	A set of filters identified to be replaced for robustness improvement.
Replacement filters	$G_\oplus$	A set of filters to replace $G_\ominus$ for robustness improvement.
Filters to be zeroized	$G_\oslash$	A set of filters, where the output of $g_i \in G_\oslash$ is a zero matrix.
Filters in remain	$G_\odot$	A subset of filters kept originally in G_i, where $G_\odot = G_i\ (G_\ominus \cup G_\oslash)$.
Sample	t	An input sample to the model under maintenance $\mathcal{F}$.
Clean dataset	T	A set of samples that do not contain any perturbations.
Noisy dataset	T^*	A set of samples, each corrupted by Gaussian Blur noise.
Spatial dataset	$T^\star$	A set of samples, each generated by spatial translation operations defined in SENSEI.
Augmentor	$\mathcal{O}$	A function to perturb the samples in T to generate T^* or $T^\star$.

To ease reference, we summarize the key symbols used in this work in Table 1.

2.1 DL Models and Convolutional Layers

A DL model $\mathcal{F}$ contains a set of layers $\langle f_1, \ldots, f_i, \ldots, f_n \rangle$. Each layer f_i accepts an input feature map and computes an output feature map. Each location in a feature map is called a neuron. The input to f_i is the output of f_{i-1}. The input layer f_1 accepts an input t of $\mathcal{F}$, and the output layer f_n produces an output o_t for $\mathcal{F}$ on processing t. Each remaining layer is a hidden layer.

$\mathcal{F}$ infers a sample t correctly if the label with the largest value in o_t is the same as the ground truth label l_t of t. The **standard accuracy** and **robustness accuracy** of $\mathcal{F}$ on a clean dataset T and a noisy dataset T^* are the proportions of samples in the corresponding datasets with correctly predicted labels, i.e., $|\{t \in T \mid \mathit{argmax}(o_t) = l_t\}|/|T|$ and $|\{t \in T^* \mid \mathit{argmax}(o_t) = l_t\}|/|T^*|$, where l_t is the ground truth label of the input t, respectively.

Convolutional layer [49] is a kind of widely used layer in the literature. As presented in Section 5 of related works, software engineering techniques using DL models with convolutional layers, such as using a sequence of ResNets in identifying invalid inputs to a DL model [51], are emerging.

Suppose f_i is a convolutional layer. It will contain a sequence of m filters $G_i = \langle g_1, \ldots, g_j, \ldots, g_m \rangle$, where m is the number of filters contained in f_i. We use the subscript j of a filter g_j to indicate the *position* of g_j in f_i, i.e., $f_i[j] = g_j$. Each **filter** g_j performs a convolution operation over the input feature map x_i of f_i to generate its output $g_j(x_i)$, called a **channel**, of the filter. The dimensions of the output channels of all filters in G_i are the same.

Two matrices A and B can be concatenated to form a larger matrix $[A\ B]$. The output $f_i(x_i)$ of a convolutional layer f_i with an input feature map x_i is the concatenation of all the channels produced by the filters of f_i. Thus, the output feature map of f_i on processing x_i can be expressed as $f_i(x_i) = [f_i[1](x_i) \ldots f_i[j](x_i) \ldots f_i[m](x_i)] = [g_1(x_i) \ldots g_j(x_i) \ldots g_m(x_i)]$. The order of channels in $f_i(x_i)$ naturally follows the positions of these filters. If the order of channels in $f_i(x_i)$ is disturbed, a different output feature map will be generated [52]. Existing studies [53–55] have shown that different filters contribute to dramatic differences in model performance.

The weights of a filter g_j is a $c \times k \times k$ matrix, called a kernel `k`. The same convolutional layer f_i contains kernels of the same size k (e.g., 3–7). The number c equals the number of channels composing the input feature map x_i for the layer f_i.

2.2 Closely related peer techniques

The DL model under maintenance $\mathcal{F}$ has been pretrained. A typical training algorithm is to solve the optimization problem of minimizing the loss between output o_t of $\mathcal{F}$ and the ground truth label l_t of each t in a given training dataset. To improve the robustness of $\mathcal{F}$, **FineTune** (FT) [56, 57] is to further train it on an adversarial dataset (containing both clean and adversarial samples). This is known as

the *adversarial training* [58], where a mixture of adversarial and clean examples is used to train the DNN model. Typically, a large set of samples with many variants are used as an adversarial dataset for finetuning. Applying FineTune with a strongly augmented dataset T°_{train} will cause a severe loss in standard accuracy [59].

In our experiments, we extensively compare DeepPatch to DeepCorrect and SENSEI. We revisit them in this section.

2.2.1 DeepCorrect

DeepCorrect [14] generates a perturbed image $\bar{t}$ from each image t in the original validation dataset T_{val} of the model under maintenance $\mathcal{F}$ to create a noisy validation dataset T^{*}_{val}. It feeds $\bar{t}$ to $\mathcal{F}$ to get the model output, thereby obtaining the robustness accuracy A on this noisy dataset. For each filter $g \in \mathcal{F}$, it feeds each $\bar{t}$ to $\mathcal{F}$, substitutes the channel of g on processing $\bar{t}$ by the channel of the same filter on processing t to get the model output, and computes the robustness accuracy B on the same set of perturbed images, thereby obtaining the robustness accuracy difference $A - B$. It prioritizes the filters in $\mathcal{F}$ in descending order of the accuracy difference. The top-k% filters are identified. DeepCorrect then revises the architecture of $\mathcal{F}$ as follows: If a layer $f_i \in \mathcal{F}$ contains any filter in this top-k% list, DeepCorect splits the original sequence of filters G_i of the layer into two subsequences, one (denoted as s_1) containing the filters appearing in the top-k% list and another (denoted as s_2) containing the remaining filters. The relative positions of the filters in either subsequence are the same as these in G_i. DeepCorrect then creates a block of two consecutive layers $\langle s_1, r \rangle$, in which r is a trainable block of layers called a correction unit [14]. It then inserts the block into f_i to replace the original sequence s_1. The output feature map of f_i in the revised $\mathcal{F}$ becomes the concatenation $[s_2(x_i)\ r(G(x_i))]$ where x_i is the input feature map of f_i. The revised $\mathcal{F}$ is then fine-tuned with only the correction units being trainable on the noisy training dataset. As such, the changes in parameter values due to learning from this noisy dataset are limited to the correction units.

2.2.2 SENSEI

SENSEI [28] is a technique based on genetic algorithm. In each retraining epoch, SENSEI evaluates whether the model under maintenance $\mathcal{F}$ is already robust to each sample t picked from the original training dataset T_{train} by translating t into 6 variants (each generated from one translation operation). If the loss values of all these variants of t are all less than a pre-defined threshold, $\mathcal{F}$ is deemed robust around t. Otherwise, the variant of t with the largest loss value is placed into a constructing robustness dataset $T^{\star}_{train}$ for that epoch. After evaluating all samples in T_{train}, it applies FineTune to retrain the current model with the robustness dataset for that epoch. SENSEI designs each variant of t as the result of applying the perturbations of each element in a chromosome on t, where each element in a chromosome corresponds to the input parameters of a translational operation (i.e., a spatial transformation operator or a sequence of such operators). Each chromosome is initialized with random parameters falling within the restrictive input ranges of the corresponding translational operations. The restrictive input range of a translation operation represents the maximum perturbation strength that can be applied to perturb a sample using the translation operation.

As we will present in Section 4, we will use the same augmented training dataset for different techniques that do not generate their augmented training dataset for robustness improvement. To conduct a controlled experiment, we produce the tool **GenRobust**, which is the same as SENSEI in all aspects except for the following. GenRobust does not evaluate whether $\mathcal{F}$ is robust to each sample $t \in T_{train}$ and does not contain the logic of only placing the variant of t with the largest loss value into the constructing spatial training dataset $T^{\star}_{train}$. Rather, it directly generates exactly one variant of t using one random chromosome and places the generated variant into the constructing the spatial augmented dataset $T^{\star}_{train}$. Since an augmentor is used in the tool, each variant is generated on the fly when $T^{\star}_{train}$ is called.

3 DeepPatch

3.1 Overview

3.1.1 Motivation and Basic Ideas

The robustness issue originates from the generalizability of the DL models on the adversarial samples with perturbations from clean samples. Recovering the filter outputs of adversarial samples to those

of respective clean samples is an intuitive way to defend the image-agnostic perturbations. Existing works [14, 60–62] follow such a method and show that recovering selective features of most sensitive filters generally outperforms fine-tuning which alters all the feature maps. DeepPatch also follows this line of research.

DeepPatch aims to equip the patched model with the full set of original filters and a selective set of replacement filters. When invoked, the replacement filters replace the functions of their original filter counterparts to compute feature maps. DeepPatch also aims to make only the replacement filters affected by the training process with adversarial examples. In this way, the perturbation strengths of adversarial examples, whether large or small, will not affect the original filters kept in the patched model. DeepPatch also shares the parameters of the original model in producing feature maps when the replacement filters are activated for processing, which alleviates the overhead in the number of parameters in the patched model.

The debugging process on traditional programs inspires us to design DeepPatch. Informally, DeepPatch first identifies a set X of trainable units in a DL model $\mathcal{F}$ that is more likely to drag $\mathcal{F}$'s leg from processing adversarial samples. This step is inspired by fault localization in debugging a traditional program to locate statements more likely to be responsible for the observed failures. Next, DeepPatch creates the little "twin brothers" of these trainable units and teaches the little brothers to process samples resembling these samples in the augmented validation dataset. Moreover, it redesigns $\mathcal{F}$ so that the redesigned model can execute either these trainable units in X or their respective "twin brothers" to process samples falling into the separation boundary of the original validation dataset side and the augmented validation dataset side, respectively. This step is inspired by repairing the code region of the located statements in debugging traditional programs, where developers tend to reuse the original code as much as possible and use the patched code to process statements for the originally incorrect situations. Finally, DeepPatch adjusts the dependency of these two kinds of units in the redesigned DL model and inserts a decision unit so that the redesigned model can automatically decide which kinds of trainable units to be executed. It thus produces the patched model at the end. This step is akin to adjusting the dependency of a program after some of its parts have been modified. These three steps are realized in DeepPatch as three tasks. They are filter assessment, model patching, and standard accuracy recovery.

3.1.2 Overall workflow

DeepPatch assumes it is in the maintenance phase of the DL model. So the data structure of the model with its trained weights and the datasets are available. The splitting of the datasets into the training, validation, and test datasets used to train the model is also known. The overall workflow of DeepPatch is as follows.

Each filter g in the original model $\mathcal{F}$ is first independently assessed on an augmented validation dataset T°_{val} with its *absence*, where the output feature map of g becomes a zero matrix. Suppose f_i is a convolutional layer in $\mathcal{F}$ and it contains a filter g. To assess the effect of g when it is absent from $\mathcal{F}$, DeepPatch first creates a patching unit α-PU for the layer f_i. This α-PU contains the filter set $G_{\oslash} = \langle g \rangle$ and an array $G_{\odot}$ of original filters of f_i but without g. It outputs the feature map consisting of the channels of the filters in $G_{\odot}$ and the zero matrix from $G_{\oslash}$, which replaces the original feature map produced by f_i. The α-PU is then inserted into the original model to create a model with one α-PU. The change in accuracy is computed by subtracting the robustness accuracy of the original model from that of the model with the α-PU on T°_{val}. All filters are prioritized into a list in descending order of their changes in accuracy on the noisy dataset.

The top-k% of the prioritized list is collected and we refer to each collected filter as a *blamed filter* and k as the *blamed ratio*. Recall that DeepPatch assesses the overall effect of each filter in the model. It could only observe whether one filter has a larger effect on robustness deterioration than another filter on the dataset representing the distribution of perturbed samples. Since original filters are trained on the original training dataset, they encode the knowledge (in the context of the model under maintenance) to process clean samples. Thus, our intention is to select a set of filters with stronger effects rather than one with weaker effects on robustness improvement to receive adversarial training on the augmented dataset. The blamed ratio is designed to specify the proportion of filters that are originally trained to infer clean samples and developers allow them to be "replaced" with replacement filters to infer samples. However, as motivated in Section 1, when the perturbation on a sample is large, these replacement filters cannot lead a model to produce accurate predictions. Thus, in our standard accuracy attention task, we present how DeepPatch addresses this problem by introducing the γ patching unit inserted into the patched model.

DeepPatch then creates a replacement filter counterpart for each blamed filter by cloning the blamed filter followed by randomly initializing the kernel of the replacement filter. Each convolutional layer $f_i \in \mathcal{F}$ containing any filter in the top-k% portion of the prioritized list will be inserted with a patching unit β-PU. Each β-PU contains three filter sets: an array $G_\ominus$ of blamed filters of f_i and its matching array $G_\oplus$ of replacement filters, and an array $G_\odot$ of filters of f_i not in $G_\ominus$, where $G_\oplus$ is initialized to $G_\ominus$. The purpose of inserting the β-PU into the layer f_i is to generate the feature map from $G_\oplus$ and $G_\odot$ to replace the original output feature map of f_i. DeepPatch patches the constructing model to replace the filter sequence in f_i with a β-PU. After patching all such layers, the model is retrained with adversarial training[2] on an augmented dataset (containing the clean and perturbed training samples) to only adjust the kernels of the replacement filters in all β-PUs for robustness improvement.

DeepPatch uses the blamed filters $G_\ominus$ and replacement filters $G_\oplus$ of the patched convolutional layer closest to the input layer of the retrained model (aka the first patched convolutional layer, denoted by f_1^{conv}) to generate a set of feature maps from each of T_{val} (the validation dataset) and T_{val}^* (the noisy validation dataset). These two sets of feature maps are used to compute a separation boundary b to distinguish them.[3] A patching unit γ-PU inherited from the first patched convolutional layer's β-PU is patched into the revised model architecture to replace the latter patching unit in the retrained model. It is configured with this boundary b. All inserted patching units are organized in a one-master-many-slaves style where the γ-PU serves as the master and all β-PU serve as the slaves. Then, DeepPatch returns the patched model.

In the forward pass of the patched model to infer a sample, when the layer f_1^{conv} is executed, the γ-PU (master) computes a value v based on its $G_\ominus$ and $G_\oplus$ on the input feature map of the layer. It compares v with b to decide whether all replacement filters in the patched model should be executed. It then instructs all patching units (all slaves and the master itself) in the patched model to act accordingly to only execute the corresponding filters to generate feature maps.

As explained in Section 1 and the above subsection, one of the main design goals of DeepPatch is to ensure DeepPatch has the *intrinsic* property of re-generating the full set of feature maps of the model under maintenance. When the patched model judges a sample falling into the original validation dataset side rather than the augmented validation dataset of the separation boundary, it produces the full set of original feature maps. We design DeepPatch to freeze all other trainable units in the above retraining step to ensure DeepPatch meets this design goal.

Alg. 1 summarizes the DeepPatch algorithm. It creates three kinds of patching units at lines 7, 14, and 22, respectively. In the following subsections, we present these patching units, a quality metric for the above-mentioned separation boundary, and the whole algorithm.

3.2 Patching Units

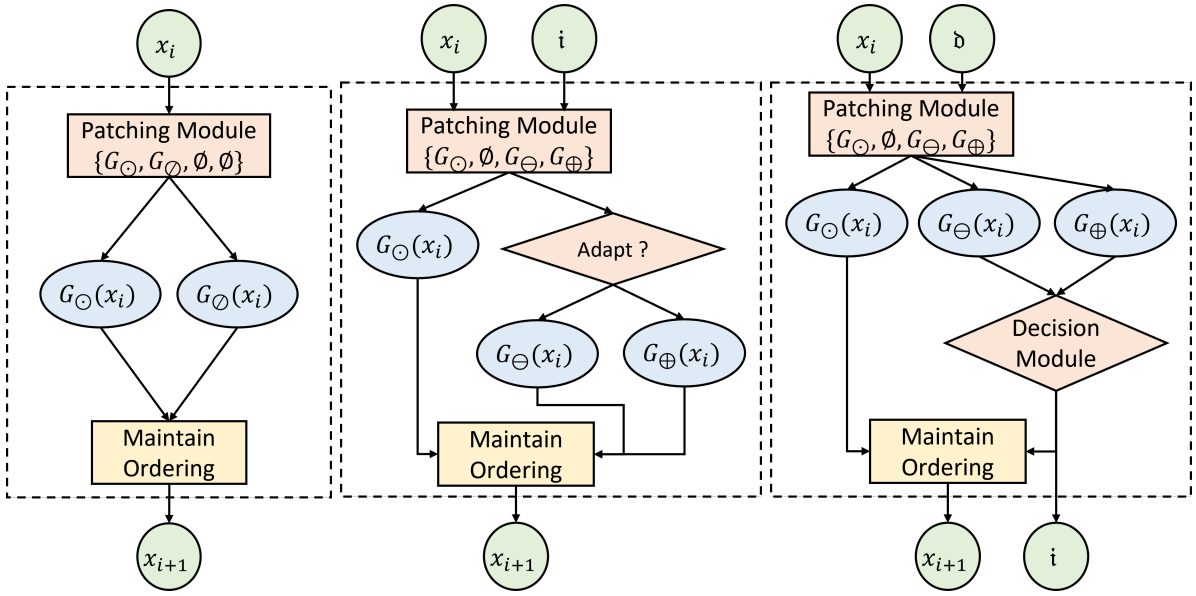
A patching unit (PU) for a convolutional layer f_i is a quadruple $PU = (G_\odot, G_\oslash, G_\ominus, G_\oplus)$, where $G_\ominus$ is a set of blamed filters, $G_\oplus$ is a set of replacement filters, $G_\oslash$ is a set of filters to be zeroized, and $G_\odot$ is the set of all the filters in f_i with the filters in $G_\ominus \cup G_\oslash$ removed. The number of filters in f_i is the same as the total number of filters in $G_\ominus \cup G_\oslash \cup G_\odot$. The number of filters in $G_\oplus$ is the same as the number of filters in $G_\ominus$. If $G_\oplus$ is nonempty, for each filter $g'_p \in G_\oplus$, there should be a matching filter $g_p \in G_\ominus$ at the same position p.

A PU also requires that each channel in its output feature map is generated by exactly one of its filters. More specifically, when a PU is called to output a feature map, the following will be output: (1) Each filter in $G_\odot$ or $G_\oslash$ will output a channel. (2) For each matching pair of filters (g_p, g'_p) where $g_p \in G_\ominus$ and $g'_p \in G_\oplus$ sharing the same position p, exactly one of them will output a channel.

The output feature map of a PU is the concatenation of the channels of the filters in the PU ordered by their filter positions. That is, $PU(x_i) = [g_{p=1}(x_i), \ldots, g_{p=j}(x_i), \ldots, g_{p=m}(x_i)]$ where x_i is the input feature map of the layer f_i, and each $g_{p=j}$ is a filter in $G_\odot \cup G_\oslash \cup G_\ominus \cup G_\oplus$. As such, the original output feature maps of a layer f_i and the PU of f_i share the same channel structure, and channels in either

[2]In adversarial training [58] for retraining, the objective is to minimize the training errors: $\text{argmin}_{\mathcal{F}} \ \mathbb{E}_{(t,l_t)\in T^\circ_{train}} \ L(\mathcal{F}, t, l_t)$, where t is the sample, l_t is the groud truth label of t, and L is the loss function to train up the pretrained model $\mathcal{F}$.

[3]In machine learning, a separation boundary between two classes in a feature space is an abstract concept, which is hard to determine precisely. In Section 3.3.2, DeepPatch projects the feature space into a single dimension. Thus, the separation boundary is a value in that dimension where all the values smaller than this boundary value represent one class, and the remaining values represent another class. We keep the terminology of *separtion boundary* in our presentation of DeepPatch to help readers to follow its high-level meaning of separating two classes, which need not be projected into a single dimension in general.



(a) α-PU (b) β-PU (c) γ-PU

Figure 2: Different kinds of Patching Unit.

output are ordered by filter positions.

3.2.1 Patching Unit α-PU

Alg. 1 at line 7 calls $\text{Patch}_\alpha(\mathcal{F}, f_i, G_\oslash)$, which creates a patching unit α-PU and patches it into $\mathcal{F}$ to create a new model $\mathcal{F}_\oslash$. The purpose of α-PU is for DeepPatch to assess the situation when the filters in $G_\oslash$ are absent from a model. Fig. 2a shows the structure of an α-PU.

In an α-PU, the two sets $G_\ominus$ and $G_\oplus$ are empty. The set $G_\odot$ is computed as the set difference $f_i \backslash G_\oslash$. An α-PU is thus $(G_\odot, G_\oslash, \emptyset, \emptyset)$. It also resets each weight in the kernel $\mathtt{k}$ of each filter $g_j \in G_\oslash$ to zero (i.e., 0) so that every filter in $G_\oslash$ will generate a zero matrix as its channel in the convolution operation during a forward pass for model prediction.

The output of an α-PU is the concatenation of the channels generated by the filters in $G_\odot$ and $G_\oslash$ ordered by their positions. Informally speaking, the output feature map of an α-PU for layer f_i is the output feature map of f_i after resetting the channels at the positions indicated by the filters in $G_\oslash$ to zeros. DeepPatch then substitutes the filter sequence kept in f_i by the α-PU to create a new model $\mathcal{F}_\oslash$.

3.2.2 Patching Unit β-PU

Alg. 1 calls $\text{Patch}_\beta(\mathcal{F}_\oplus, f_i, G_\oplus, G_\ominus)$ at line 14 to create a β-PU. The purpose of β-PU is to produce a perturbation-adaptive output feature map for selected channels (indicated by the filters in $G_\ominus$) of the layer f_i. In a $\beta\text{-PU} = (G_\odot, \emptyset, G_\ominus, G_\oplus)$, the set $G_\oslash$ is empty, and $G_\odot$ is computed as the set difference $f_i \backslash G_\ominus$. Fig. 2b shows the structure of a β-PU.

Each β-PU shares the same *adaptation indicator* $\mathfrak{i}$ (a Boolean variable with TRUE as the default value) for selecting either $G_\ominus$ or $G_\oplus$ for computing the output of the β-PU. If the value of $\mathfrak{i}$ is FALSE, the β-PU computes the channels from $G_\odot$ and $G_\ominus$ only (i.e., without any output from the filters in $G_\oplus$). So, it generates an output not affected by the filters in $G_\oplus$. On the contrary, if $\mathfrak{i}$ is TRUE, the β-PU computes the channels from $G_\odot$ and $G_\oplus$ only, indicating that it intends to deem the current sample under processing more like an adversarial sample. The default value is chosen as TRUE because the algorithm at line 18 trains all replacement filters in all β-PUs on both the original and perturbed samples. The replacement filters in $G_\oplus$ are trained with adversarial training because they are more likely than the blamed filters in $G_\ominus$ to generalize over the original and perturbed samples.

DeepPatch then replaces the filter sequence kept in f_i of the model $\mathcal{F}$ by this β-PU to evolve $\mathcal{F}_\oplus$ (line 14 of Alg. 1). It repeats the process for all convolutional layers. It retrains the resultant $\mathcal{F}_\oplus$ model using adversarial training [58] (line 18). Note that at this moment, all β-PUs have a default value for $\mathfrak{i}$ with TRUE and only set the filters in $G_\oplus$ to be trainable. Thus, in the training process, each β-PU uses

$G_\odot$ and $G_\oplus$ to compute its output feature map and update the kernel weights in $G_\oplus$ only.

3.2.3 Patching Unit γ-PU

Alg. 1 at line 22 calls $\text{Patch}_\gamma(\mathcal{F}_\oplus, f_i, G_\oplus, G_\ominus, b)$. γ-PU contains the same sets of filters as the β-PU at the same layer f_i with an additional input parameter b (a separation boundary). In DeepPatch, we design to attach a γ-PU to the first patched convolutional layer f_1^{conv} (see line 22 of Alg. 1) to act as the master in the master-and-slaves style. Fig. 2c depicts the structure of a γ-PU structure. The reason for attaching this patching unit to f_1^{conv} is to allow all convolutional layers in the patched model to use their replacement filters for feature map computations. For instance, consider a patched version of the ResNet32 model for the CIFAR10 dataset. Suppose a γ-PU is attached to the last convolutional layer of the patched model instead of the first one. In this case, only the replacement filters in the last convolutional layer can be used for predicting the output for a sample. In a convolutional neural network, a preceding convolutional layer generates relatively lower-level features (e.g., leg, hair) and a subsequent convolutional layer generates relatively higher-level features (e.g., a region with four legs). Making the decision of choosing which set of filters to be activated in a preceding layer can make a large set of the original or replacement filters capture different levels of features for processing a sample. To avoid losing some features at the low level when deciding to activate or not to activate all replacement filters, DeepPatch chooses to add the γ-PU to the first convolutional layer.

A γ-PU accepts an indicator $\mathfrak{d}$ as well, which is *decision indicator* (a Boolean variable with TRUE as the default value), indicating whether to enable the decision module or not. Note that this decision indicator can be set by the user manually or omit it to be the default value. If the value of $\mathfrak{d}$ is FALSE, the γ-PU will set the adaptation indicator $\mathfrak{i}$ to FALSE and compute the channels from $G_\odot$ and $G_\ominus$ to produce the feature map. If the value of $\mathfrak{d}$ is TRUE, when $\mathcal{F}_\oplus$ conducts an inference on an input t, the γ-PU computes the channels of the filters in $G_\ominus$, denoted by z^-, and those in $G_\oplus$, denoted by z^+. It calls the function *Calibrate*(z^-, z^+) to return a value, say v. The γ-PU then computes whether the condition $v > b$ holds (where b is computed by line 21 of Alg. 1 at the maintenance phase). If the condition holds, the adaptation indicator $\mathfrak{i}$ is set to TRUE, otherwise FALSE, which will be passed to all the β-PUs.

Like a β-PU, the γ-PU also generates an output feature map. If the adaptation indicator is FALSE, the γ-PU computes the channels from $G_\odot$ and $G_\ominus$ only (producing the feature maps of $\mathcal{F}$), otherwise from $G_\odot$ and $G_\oplus$ only. After line 22, the current model has been patched with a γ-PU at the first convolutional layer.

Our design of deciding the activation of different filters for different kinds of inputs is novel, and we are unaware of any existing work having a similar design. Another novel design is the support of the reproduction of the original feature maps after training with adversarial examples.

3.3 Robustness-Aware Quality Metric as Separation Boundary

The γ-PU requires a separation boundary in the final DL model. A constraint is that we have to design a scheme that should be efficient in model prediction.

3.3.1 Corrective Rate

Our key insight is that the two sets of filters in $G_\ominus$ and $G_\oplus$ lead the final DL model to behave differently when handling samples more like the augmented validation dataset (than the original validation dataset) but perform more similarly on samples more like the original validation dataset (than the augmented validation dataset). In the literature, the existing work (e.g., [60]) shows that the neurons previously not activated in the pretrained models become activated in the retrained models, resulting in a large robustness accuracy improvement compared to the pretrained models. In our experiment (Tables 2 versus 5), we also observe that the pretrained models and the patched models produced by DeepPatch with β-PUs have a larger difference in robust accuracy than that in standard accuracy. As such, we tend to consider the filters in $G_\oplus$ can serve like the original filters to some degree in inferring samples more like the original validation dataset. There are many possible formulations to implement the insight. However, since the computation should be performed while the model is being used for prediction, it should be lightweight and easy to compute.

We recall that the convolution operations done by the filters in $G_\ominus$ and $G_\oplus$ produce two concatenations of their channels, respectively. We denote these two outputs $G_\ominus(x)$ and $G_\oplus(x)$ by z^- and z^+, respectively, where x is the input feature map. Note that both z^- and z^+ are feature maps produced

from the activation functions of the layer[4]. DeepPatch measures the differences between these two feature maps at the first convolutional layer f_1^{conv} in the patched model at the neuron pair level. In the task of standard accuracy recovery, it uses all these differences on the clean and augmented validation datasets to calibrate the separation boundary.

We propose *Corrective Rate* (CR), which measures on two feature maps z^- and z^+ computed in γ-PU. We refer to the neuron in the same positions in a pair of channels as activated but misaligned if the neuron is only activated in exactly one of the two channels. CR computes the ratio of the number of activated but misaligned neurons between the channels of blamed and replacement filters to the number of all neurons produced by the replacement filters. As explained earlier in this section, if the input sample is more like the augmented validation dataset than the clean validation dataset, the activated neurons tend to get a larger misalignment between $G_\ominus$ and $G_\oplus$.

$$\mathrm{CR}(z^-, z^+) = \frac{|\{\eta \in z | z = xor(\diamond(z^-), \diamond(z^+)) \wedge \eta = 1\}|}{|\{\eta \in z^+\}|} \tag{1}$$

where z^- and z^+ are defined above, η denotes a neuron in the feature map z. $\diamond(\eta)$ is the Heaviside step function [63] that outputs 1 if $\eta > 0$, and 0 otherwise. $xor(\cdot)$ is the logical exclusive-or function over $\diamond(z^-)$ and $\diamond(z^+)$, where for each corresponding pair of neurons in $\diamond(z^-)$ and $\diamond(z^+)$, it behaves: xor(1,0) = 1, xor(0,1) = 1, xor(0,0) = 0, and xor(1,1) = 0, and z is the output of $xor(\cdot)$ on all neurons.

The idea of calculating the activated and misaligned neurons is to estimate the extent of feature map similarity if $G_\oplus$ is used to generate the feature map of the layer instead of $G_\ominus$.

For example, suppose $z^- = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ and $z^+ = \left(\begin{smallmatrix} 1 & 1 \\ 1 & 1 \end{smallmatrix}\right)$, then the computed z is $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. The numerator of the equation is 3 by counting the numbers of 1, and the denominator is 4 defined by the shape of z^+. The final result of CR is 0.75 in this case, indicating that 75% of the neurons on the feature maps get *corrected* if z^+ is used to generate the feature map for the layer.

3.3.2 Calibration function

Line 21 of Alg. 1 requires a calibration function to compute a separation boundary b. The function computes the value of the CR metric for each sample in each of the original validation dataset T_{val} and the augmented validation dataset T_{val}°. This process creates two sets of values. For either set of values, the cluster center is computed. Since CR computes a scalar value, the cluster center is the mean value of the values in the set. The function then returns the mean value of these two cluster centers as the separation boundary b, which is used by the γ-PU.

3.4 Algorithm

Alg. 1 shows the DeepPatch algorithm. It accepts five inputs: a DL model $\mathcal{F}$, a blamed ratio $\kappa \in (0, 100]$, an augmentor $\mathcal{O}$ to perturb the samples, and a pair of datasets (T_{train} and T_{val}) for training a model for robustness improvement.

There are three tasks in this algorithm: filter assessment, model patching, and standard accuracy recovery. To better demonstrate how the algorithm involves the three kinds of PUs in the model under maintenance, we provide a step-by-step explanation of these three tasks.

3.4.1 Filter assessment task

At line 1, the robustness accuracy of the given model $\mathcal{F}$ is measured as the base robustness accuracy acc_0 by running the inference process (denoted by $Inference(\cdot)$) *on the augmented validation dataset* T_{val}°. The algorithm then iterates on each convolutional layer f_i to assess the filters.

The filter assessment task assesses the impact on robustness accuracy at the individual filter level. It iterates over all the filters of the layer f_i (lines 5–10). In each iteration, say for filter g_j, it creates a patched version of $\mathcal{F}$ inserted with a patching unit α-PU, through the function $\mathsf{Patch}_\alpha(\cdot)$ (lines 6–7). The purpose of the patching unit α-PU is to nullify the channel generated by g_i in the output feature map of f_i (i.e., the channel is zeroized). Subsection 3.2 has presented the procedure. Then, the algorithm evaluates $\mathcal{F}_\oslash$ over the augmented samples (T_{val}^*) to obtain the robustness accuracy acc of the model, which is further compared with the base robustness accuracy to compute a difference ($\delta_j = acc - acc_0$) for the filter g_j. Among the filters of the same layer, the algorithm deems a filter g_j with a larger difference

[4]A feature map can be viewed as a sequence of values. The sequence keeps a value at each index position. The index position of the sequence is called a neuron.

Algorithm 1: Patching a model with Patching Unit

Input : Pretrained model $\mathcal{F}$,
Blamed ratio κ,
Augmentor $\mathcal{O}$,
Training set T_{train},
Validation set T_{val}

Output: Patched model $\mathcal{F}_{\oplus}$

1 $acc_0 \leftarrow \mathit{Inference}(\mathcal{F}, T^{\circ}_{val})$; // augmented by $\mathcal{O}$ on T_{val}, same below
2 $\mathcal{F}_{\oplus} \leftarrow \mathcal{F}$
3 **foreach** *convolutional layer* $f_i \in \mathcal{F}$ **do**
4 $\quad Q \leftarrow$ empty priority queue
5 $\quad$ **foreach** *filter* $g_j \in f_i$ **do**
6 $\quad\quad G_{\oslash} \leftarrow \{g_j\}$
7 $\quad\quad \mathcal{F}_{\oslash} \leftarrow \mathit{Patch}_{\alpha}(\mathcal{F}, f_i, G_{\oslash})$
8 $\quad\quad acc \leftarrow \mathit{Inference}(\mathcal{F}_{\oslash}, T^{\circ}_{val})$
9 $\quad\quad Q \Leftarrow (g_j, acc - acc_0)$
10 $\quad$ **end foreach**
11 $\quad G_{\ominus} \leftarrow$ top $\kappa\%$ in Q
12 $\quad \mathtt{K} \leftarrow \mathit{extract}(G_{\ominus})$
13 $\quad G_{\oplus} \leftarrow \mathit{reinitialize}(\mathtt{K})$
14 $\quad \mathcal{F}_{\oplus} \leftarrow \mathit{Patch}_{\beta}(\mathcal{F}_{\oplus}, f_i, G_{\oplus}, G_{\ominus})$
15 **end foreach**
16 $\mathit{freeze}(\mathcal{F}_{\oplus})$
17 **forall** $G_{\oplus} \in f_i \in \mathcal{F}_{\oplus}$ **do** $\mathit{trainable}(G_{\oplus})$;
18 $\mathcal{F}_{\oplus} \leftarrow \mathit{Train}(\mathcal{F}_{\oplus}, T^{\circ}_{train}, T^{\circ}_{val})$; // augmented by $\mathcal{O}$ on T_{train}
19 $f_1^{conv} \leftarrow$ first convolutional layer of $\mathcal{F}$
20 $G_{1\oplus}, G_{1\ominus} \leftarrow \mathit{Detach}(\mathcal{F}_{\oplus}, f_1^{conv})$
21 $b \leftarrow \mathit{Calibrate}(T_{val}, T^{\circ}_{val}, G_{1\oplus}, G_{1\ominus})$
22 $\mathcal{F}_{\oplus} \leftarrow \mathit{Patch}_{\gamma}(\mathcal{F}_{\oplus}, f_1^{conv}, G_{1\oplus}, G_{1\ominus}, b)$

δ_j to have a higher potential to improve the robustness of $\mathcal{F}$ than the other filter of the same layer with a smaller difference. The pair (g_j, δ_j) is added into the priority queue Q (reset to empty at line 4), where g_j is the element to be prioritized in the queue, and δ_j is the priority score of g_j (line 9). The queue Q thus prioritizes the filters of the same layer in descending order of priority score. Note that the priority queue Q is initialized as empty at line 4 when processing each iteration on the set of convolutional layers in $\mathcal{F}$.

3.4.2 Model patching

The second task first creates a set of replacement filters for f_i (lines 11–13). The algorithm identifies the top-κ% filters in the priority queue Q for the layer at line 11. We denote the list of top-κ% filters by the sequence $G_{\ominus}$ (where $G_{\ominus}$ is the prefix of Q such that $|G_{\ominus}| = \lfloor \kappa |Q| \rfloor$). The algorithm then creates a set of kernels of the replacement filters that will be patched into the cloned version of $\mathcal{F}$, where such $\mathcal{F}$ is created at line 2 and incrementally patched on line 14 in the loop. In line 12, it identifies the dimension of each such kernel (because each filter has its own dimension). The function $\mathit{extract}(\cdot)$ firstly creates $\mathtt{K} = \langle\ \rangle$. For each filter g_j in $G_{\ominus}$, it finds the kernel k_j of the filter g_j from the given model $\mathcal{F}$, and appends k_j to $\mathtt{K}$. Thus, the algorithm obtains a sequence of kernels $\mathtt{K} = \langle k_1, \ldots, k_j, \ldots, k_{|G_{\ominus}|} \rangle$, where k_j is for $g_j \in G_{\ominus}$ for all $1 \leq j \leq |G_{\ominus}|$.

In line 13, the algorithm randomly reinitializes all kernels in $\mathtt{K}$ to create a new set of filters $G_{\oplus}$ via $\mathit{reinitialize}(\cdot)$, one new filter for each blamed filter in $G_{\ominus}$. Every matching pair of filters between the two sets $G_{\oplus}$ and $G_{\ominus}$ has the same position index in the layer f_i. The purpose of creating $G_{\oplus}$ is to prepare $\mathcal{F}$ to process samples more like the augmented validation dataset (see below).

The algorithm then constructs a patching unit β-PU for the layer f_i using the two sets of filters $G_{\oplus}$ and $G_{\ominus}$, and patches the model $\mathcal{F}_{\oplus}$ at line 14. Subsection 3.2 has presented how a β-PU is constructed. The newly reinitialized trainable weights in $\mathcal{F}_{\oplus}$ require retraining. The algorithm freezes all the weights in $\mathcal{F}_{\oplus}$ via the function $\mathit{freeze}(\cdot)$ (line 16). Then, it unfreezes the kernel of each replacement filter in each β-PU patched into $\mathcal{F}_{\oplus}$ via the function $\mathit{trainable}(\cdot)$ (line 17).

At line 18, the partially unfrozen model $\mathcal{F}_{\oplus}$ is trained with adversarial training [58] on both the clean and perturbed samples (i.e., over T°_{train} and T°_{val}) to train the kernels of the replacement filters. After this retraining, the weights in $G_{\oplus}$ have been adjusted, and the remaining weights in $\mathcal{F}_{\oplus}$ are kept unchanged (i.e., same as the corresponding weights in $\mathcal{F}$).

3.4.3 Standard accuracy recovery

We aim to reproduce the original feature maps of $\mathcal{F}$ on demand when predicting a sample that resembles more the original validation dataset than the augmented validation dataset. Since each β-PU has both original and replacement filters, this task is to configure each convolutional layer to linearly separate (for efficient computation reasons) individual cases to be processed by the original filters or by the replacement filters so that any full set of feature maps produced by $\mathcal{F}$ can be reproducible by $\mathcal{F}_{\oplus}$ when needed.

In lines 19–20, the algorithm firstly detaches all newly added filters $G_{1\oplus}$ and all blamed filters $G_{1\ominus}$ from the first patched convolutional layer f_1^{conv} of the retrained model $\mathcal{F}_{\oplus}$. It then treats all the filters in each of $G_{1\oplus}$ and $G_{1\ominus}$ as a DL model with a single hidden layer (where the filters lie) to transform each input t into two matrices. The transformation is applied to each input in the original and augmented validation datasets T_{val} and T°_{val} (line 21), producing four sets of matrices, one for each combination. Then, it fits these matrices to compute a boundary to separate the two matrices for T_{val} from the two matrices for T°_{val}, which we refer to it as the separation boundary b. The above transformation and boundary computation are done via the function *Calibrate*$(\cdot)$ (see Subsection 3.3). After computing a separation boundary b, DeepPatch inserts the patching unit γ-PU with the separation boundary b (line 22) into f_1^{conv} of the retrained model $\mathcal{F}_{\oplus}$, and the original β-PU in the same layer has been detached at line 20. So, f_1^{conv} has only one PU after line 22. The algorithm then returns the resultant $\mathcal{F}_{\oplus}$.

We note that DL model layers serve as a sequence of transformations, and the enabled filters must be consistent with the forward pass in the training time for the model prediction to be effective. Thus, DeepPatch organizes all the patching units in a one-master-many-slaves design style to ensure that filters are activated consistently, where the γ-PU serves as the master and all other patching units serve as slaves.

When the resultant model $\mathcal{F}_{\oplus}$ is used for inferring an input, the γ-PU within this model will decide whether the model uses its blamed filters to generate channels for the output feature map of the layer with γ-PU, and if not, it uses the original filters instead. In the program code of the patched model, the master patching unit (γ-PU) updates a program variable called adaptation indicator, and all patching units will read this variable and activate their respective filters consistently on processing their input feature maps.

3.5 Code Changes in Architecture Program

Apart from modifying the model weights $\mathcal{F}$ through retraining, DeepPatch modifies the model architecture by inserting code into the model architecture program (see Listing 1 for illustration).

In Listing 1, there are two sections. Lines 50–53 show a fragment of an exemplified model with a convolutional layer. To patch the model with a β-PU to this convolutional layer, DeepPatch locates the Conv2d instantiation (line 52) in the program script and then replaces it with the instantiation of a β-PU (line 53), where the original Conv2d instance is wrapped into the β-PU, and the second parameter (e.g., [0, 3, ...]) is the filter indices specified for replacement filters, which is automatically generated by the DeepPatch tool. The DeepPatch tool will scan through the model architecture to locate all the convolutional layers and dynamically substitute them with β-PUs. The patching of the model architecture with α-PU and γ-PU is similar, except that for γ-PU, it also has the code of functions (lines 31–37) to compute our quality metric value (i.e., CR value) and compare the metric value with the separation boundary b.

Then, the DeepPatch tool performs the respective tasks according to different kinds of patching units. For example, in the standard accuracy recovery task, the execution flow of γ-PU is depicted in Section 3.2.3. In the *forward* function, the variable x and a variable representing the decision indicator are input for inference. The γ-PU firstly determines whether it should make a decision by itself according to the decision indicator. If so, the γ-PU computes the corrective rate and then compares it to the separation boundary to make a comparison with another variable representing the adaptation indicator. If not, the γ-PU sets the adaptation indicator to FALSE. The value of the adaptation indicator is read by all β-PUs for controlling the program execution. This design distinguishes DeepPatch from other closely related techniques, such as DeepCorrect [14], SENSEI [28], RobOT [48], and DeepRepair [19].

```
# Maintenance modification to the model architecture
class AlphaUnit(nn.Module):
    def __init__(self, conv_layer, indices):
        self.conv = conv_layer
        self.indices = indices

    def forward(self, x):
        out = self.conv(x)
        out[self.indices] = 0
        return out

class BetaUnit(nn.Module):
    def __init__(self, conv_layer, indices):
        self.conv = conv_layer
        self.repl = nn.Conv2d(conv_layer.in_channels, len(indices))
        self.indices = indices

    def forward(self, x, adaptation_indicator=True):
        out = self.conv(x)
        if adaptation_indicator is True:
            out[self.indices] = self.repl(x)
        return out, adaptation_indicator

class GammaUnit(nn.Module):
    def __init__(self, conv_layer, indices, b):
        self.conv = conv_layer
        self.repl = nn.Conv2d(conv_layer.in_channels, len(indices))
        self.indices = indices
        self.boundary = b

    def corrective_rate(self, f1, f2):
        # code skipped. See Section 3.3.1
        return CR

    def make_decision(self, f1, f2):
        cr = self.corrective_rate(f1, f2)
        return cr > self.boundary  # True of False

    def forward(self, x, decision_indicator=True):
        out = self.conv(x)
        if decision_indicator is True:
            counterpart = self.repl(x)
            adaptation_indicator = self.make_decision(out[self.indices], counterpart)
            if adaptation_indicator is True:
                out[self.indices] = counterpart
        else:
            adaptation_indicator = False
        return out, adaptation_indicator

class ExampleModel(nn.Module):
    def __init__(self):
        # self.conv = nn.Conv2d(3, 5)                       # original
        self.conv = BetaUnit(nn.Conv2d(3, 5), [0, 3, ...])  # patching example
```

Listing 1: Example patching code changes to a model architecture

Finally, the resultant patched model is saved into files with the help of the platform API (e.g., PyTorch). The files contain the model architecture and weights, which can be reloaded back to the program with the help of the platform API for the next usage. The patching process and the execution of all the tasks of all kinds of patching units are fully automated. DeepPatch is an end-to-end tool and is open-source [64] for reference and practical use. A prototype of DeepPatch is also integrated into the SEBox4DL tool [65], which provides an easy-to-use UI and workflow automation.

4 Evaluation

4.1 Research Questions

We aim to answer the following four research questions.

- RQ1: How effectively does DeepPatch perform in patching DL models for effective robustness improvement with retention of the original standard accuracy?
- RQ2: What are the effects of different DeepPatch variants on maintaining the models?
- RQ3: What are the effects of using DeepPatch to improve SENSEI-generated models and using SENSEI to improve DeepPatch-generated filters?
- RQ4: What are the effects of the blamed ratio in robustness improvement? Does the filter prioritization strategy used in DeepPatch produce consistent effects on standard and robustness accuracy across models and across different perturbation strengths?

4.2 Experimental Setup

4.2.1 Implementation and Environment

We implemented our experiment framework in Python v3.8 and Pytorch v1.8.1 [66]. We ran the experiments on a Ubuntu 20.04 server with a 48-core 3.0GHz Xeon CPU, 256GB RAM, and a 2080Ti GPU. DeepPatch was implemented as a tool used in the framework, with the help of a public toolkit SEBox4DL [65]. The CR metric was computed by matrix operations in GPU. The framework, the tool, and all code in the experiments are available at Github [64].

For each evaluation of a single technique, a general process was to first retrain the model by using the algorithm described in the technique and then evaluate the performance of the retrained model on a series of test datasets. In the retraining step, we used stochastic gradient descent (SGD) [67] as the optimizer with the parameters of the learning rate of 0.01, the momentum of 0.9, and the weight decay of $5e^{-4}$ for the training algorithm. Cosine annealing scheduler [68] with parameter T_max = 200 was applied as the learning rate scheduler. The training dataset contains both clean samples and perturbed samples, following the adversarial training process in [58]. We retrained the models with 50 epochs and ensured that the retraining caused the models to have converged.

In DeepPatch, the number used to select a certain percentage of filters for model patching was rounded down to the nearest integer in all experiments. For all techniques, if the retraining process cannot produce a model with a higher validation accuracy than the pretrained model, which means that the highest validation accuracy is achieved by the model state before retraining, then the pretrained model is produced as the retraining output.

4.2.2 Peer Techniques

We compared DeepPatch (DP) with a diverse set of peer techniques to evaluate its effectiveness. They were **DeepCorrect** (**DC**) [14], **SENSEI** (**SS**) [28], **Apricot** (**AP**) [15], **DeepGini** (**DG**) [25], and **AugMix** (**AM**) [69]. They aim at robustness improvement. The pipeline DC+FT is to serve as a baseline that improves the robustness followed by improving the standard accuracy. We also compared DeepPatch with the original **Pretrained** model (**PT**) of each benchmark.

Both DeepCorrect and SENSEI were configured to evaluate adversarial examples with diverse, both small and large, perturbation strengths in their original experiments [14, 28]. We adopted their settings of perturbation strengths and integrated all peer techniques into our framework.

DC DeepCorrect [14] was the state-of-the-art technique that modified filters for robustness improvement of Gaussian blur noise and revised architecture programs. We have revisited DeepCorrect in Section 2.2.1. DeepPatch was heavily inspired by DeepCorrect, and thus we chose to compare DeepPatch with it.

SS SENSEI [28] was the state-of-the-art fuzzing technique to select the training samples to train the model for spatial robustness. We have revisited SENSEI in Section 2.2.2. We compared DeepPatch with SENSEI to evaluate whether the design of DeepPatch can be generalized to retain the standard accuracy while handling a new type of noise (spatial robustness) beyond its original task inspired by DeepCorrect.

FT FineTune [70] is a state-of-the-practice technique to retrain a model on a dataset until convergence. The previous experiments have shown that DeepCorrect outperformed FT [14]. FT trained a model on the expanded dataset containing all the samples in the original training dataset and the augmented training dataset using the original loss function that produced the model. We included FT to explore the possibility of recovering the loss in standard accuracy introduced by DeepCorrect. We also note that SENSEI and Apricot have incorporated FT in their internal training process.

DC+FT This pipeline is DeepCorrect followed by FineTune (DC+FT). Our idea for formulating DC+FT is motivated by the close relationship between DeepCorrect and DeepPatch. In this pipeline, the output model produced by DC is further finetuned by FT on the original (clean) training dataset. We treat the pipeline as a variant of DeepCorrect so that the variant has a component for robustness improvement followed by a component for standard accuracy recovery. Using this variant in the experiment increases the alignment between DeepCorrect and DeepPatch in the experiment.

AP Apricot [15] was a state-of-the-art technique to boost the standard accuracy using a weight-adaptation approach. It created a set of submodels of an original model. It then iteratively transfers the aggregated weights of the submodels that infer samples correctly and incorrectly (through an aggregation strategy) predicted by the constructing model (where the model is first initialized with the original model). AP adopted Strategy 2 of the original Apricot paper as the aggregation strategy because this strategy was more effective than other strategies in the original experiment. We applied AP to the robustness improvement task.

DG DeepGini [25] was a metric-based state-of-the-art test case selection technique for robustness improvement. It proposed to use the Gini impurity index to evaluate each sample, rank samples in descending order of the Gini index value, and select the top k% of ranked samples based on this index for retraining. We provided the adversarial training dataset to DG for its selection to produce a retraining dataset followed by applying FT to retrain the pretrained model on this retraining dataset.

RB RobOT [48] proposed a metric called the first-order loss, which measured the Euclidean norm of a sample. It proposed a number of techniques in the paper. Among them, there were two test case selection methods, KM-ST and BE-ST. The experimental results presented in Table IV of the original RobOT paper showed that BE-ST was more effective than KM-ST. BE-ST reordered the samples in a given dataset in descending order of the first-order loss and selected an equal number of samples from both ends of the reordered list. We used the BE-ST method to stand for RB in our experiment. Like DG above, RB selected k% of the adversarial training dataset to produce a retraining dataset followed by retraining the model on this retraining dataset by FT.

AM AugMix [69] was a well-known, simple, and efficient data augmentation technique in the machine learning community to jointly improve robustness and uncertainty. AugMix introduced the Jensen-Shannon divergence consistency loss [71] as a regularization for adversarial training among augmented samples. We compared DeepPatch with AugMix to evaluate how DeepPatch compares to methods used for data augmentation in the real world.

4.2.3 Models

We aim to explore more and diverse kinds of CNN model architectures within our limited computational resources. Six publicly available and pretrained models with high test accuracy were selected from popular GitHub repositories in the experiments. ResNet32 [41] and MobileNet [72] were pretrained on CIFAR10, and VGG [73] and ShuffleNetV2 [74] were pretrained on CIFAR100. ResNet18 [41] was

pretrained on TinyImageNet, and a ResNet-like CNN model was pretrained on SVHN. Note that since CIFAR10 and CIFAR100 shared the same input dimension and image quality but with a different number of output classes, we configured more dataset and model combinations for them. All the pretrained models and their architecture programs and weights were downloaded from [75–77]. We evaluated them on their respective test datasets (T_{test}) to confirm the pretrained performance and find out their robustness accuracy in our experiment settings. The standard accuracy of these pretrained models matched the published benchmark values [78]. Table 2 shows the descriptive statistics of the datasets and models.

Table 2: Descriptive Statistics on Datasets and Models

Index	Dataset	Model	#Conv	#Filters	#Params	std(%)	nRob(%)	sRob(%)
①	CIFAR10	ResNet32	33	1232	466,906	93.53	42.29	38.95
②	CIFAR10	MobileNetV2	52	9272	700,490	93.12	42.33	37.85
③	CIFAR100	VGG13	10	2944	9,987,492	74.63	27.14	18.91
④	CIFAR100	ShuffleNetV2	56	8090	1,356,104	72.63	27.11	17.69
⑤	STL10	Res-CNN	6	864	1,128,650	77.58	48.06	32.58
⑥	TinyImageNet	ResNet18	20	4800	11,271,496	72.72	30.63	61.46

* Conv, Filters, and Params denotes all Filters in Convolutional layers with trainable Parameters, respectively. std for standard accuracy, nRob for noisy robustness accuracy, sRob for spatial robustness accuracy, and joint for joint accuracy.

4.2.4 Datasets

We adopted CIFAR10 [42], CIFAR100 [79], STL10 [80], and TinyImageNet [81] datasets as our benchmark datasets. These datasets cover diverse dimensions, such as the number of classes (including 10, 100, and 200 classes), input image sizes (including 32x32, 64x64, and 96x96 pixels), and the number of samples (including 13000, 60000, and 110000 samples). To retrain such a model, the downloaded datasets are split into the training dataset, validation dataset, and test dataset, described in the following clean dataset construction subsection. To evaluate the noisy and spatial robustness accuracy, two kinds of augmented datasets are constructed, which are introduced in the noisy dataset construction subsection and spatial dataset construction subsection, respectively. These constructed datasets contained diverse samples with different levels of perturbation strength, and they were used as the augmented datasets (i.e., T°_{train} and T°_{val}) in Alg. 1.

(a) Clean dataset construction. Each benchmark dataset had an original training dataset $T_{original}$ and a test dataset T_{test}. We divided $T_{original}$ into two parts using the *train_test_split* (in the Python sklearn package `https://scikit-learn.org/`) with a ratio 1:49 to split it into a validation dataset T_{val} and a training dataset T_{train} in our experiment. A constant seed ($s = 2022$) was used to make the splitting consistent, and a parameter *stratify* was set to sample labels to make the class balanced in both split sets. The original test dataset T_{test} is used as the test dataset in our experiments.

(b) Noisy dataset construction. We adopted and ran the tool of DeepCorrect [14] to construct noisy datasets T^{*}_{train} with the Gaussian Blur [82] noise type from the samples in the corresponding clean datasets. The Gaussian Blur noise was implemented as an augmentor $\mathcal{O}_1$ used in Alg. 1. Following the original experiment in DeepCorrect [14], we configured the DeepCorrect tool with six levels of noise for perturbation with standard deviation σ in the set $\Sigma = \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0\}$. The kernel size λ in Gaussian Blur was set according to the chosen σ value, i.e., $\lambda = \lceil 4 * \sigma \rceil + 1$.

The tool generated adversarial training samples in the following dynamic manner:

$$\text{yield}(t, blur, \Sigma) = \begin{cases} t, & \text{if } rand(\{0,1\}) \text{ is } 0. \\ blur(t, rand(\Sigma)), & \text{otherwise.} \end{cases} \tag{2}$$

where t is an input clean training sample, *blur* is the Gaussian Blur operation to perturb the sample, and Σ is the set of parameters used in *blur*. In each training epoch when retraining a model, and for each sample t in T_{train}, the tool called the $yield(\cdot)$ function to randomly decide whether t was to be perturbed. If t was chosen to be perturbed, the tool randomly picked one value from the set Σ, applied the perturbation operation with this value as the perturbation strength on t to generate a perturbed sample $\bar{t}$ and used $\bar{t}$ for retraining in the current epoch. Note that in different epochs, the yielded training samples may not be identical due to the dynamic manner, which is a typical practice of adversarial training [38] and is widely used in many robustness improvement works [28, 48].

The tool generated adversarial samples for validation and testing in a static manner. For each combination of $t \in T_{test}$ and $\sigma \in \Sigma$, it applied the perturbation operation with the σ value as the

perturbation strength on t to generate a perturbed sample $\bar{t}$ and place $\bar{t}$ into the set $T^{*\sigma}_{test}$. The set $T^{*}_{test} = \bigcup_{\sigma \in \Sigma} T^{*\sigma}_{test} \cup T_{test}$ was constructed as the noisy test dataset. The same procedure for constructing the noisy validation dataset T^{*}_{val} followed the construction process of T^{*}_{test} except that the tool used T_{val} instead of T_{test} as its input, and T^{*}_{val} does not contain samples in T_{val}. Note that there was only one fixed T^{*}_{val} with the size six folds to the size of T_{val} for the whole training process.

(c) Spatial dataset construction. We first ported the SENSEI tool [83] to our platform owing to the difference in the underlying platforms. The original SENSEI tool chose a perturbation bound τ for each translation operation. For instance, the rotation translation in SNESEI for an image sample accepted a value within the restrictive input range $[-\tau, \tau]$ to rotate the image by the angle specified by the value. The value represented a perturbation strength applied to the image sample. The range $[-\tau, \tau]$ represented the level of perturbation strength. We refer to it as the perturbation range for the translation operation.

We then created the GenRobust tool from the SENSEI tool according to the description presented in Section 2.2.1. In brief, the GenRobust tool was the same as the SENSEI tool except that it randomly generated exactly one perturbed sample from a given sample with a perturbation strength falling within the original perturbation range. The tool was also used as an augmentor $\mathcal{O}_2$ in Alg. 1. To evaluate a sample, the GenRobust tool was used to generate 50 distinct perturbed samples of the sample for each translation operation. This is because Table 5 in the SENSEI original paper [28] showed that using 50 as the value for this hyperparameter was sufficient to estimate the spatial robustness measurement for a sample.

We ran the GenRobust tool to construct a spatial training dataset $T^{\star}_{train}$, a spatial validation dataset $T^{\star}_{val}$, and a spatial test dataset $T^{\star}_{test}$ from the training, validation, and test datasets of each benchmark (i.e., T_{train}, T_{val}, T_{test}), respectively. Similar to the noisy datasets, $T^{\star}_{train}$ dynamically generated different training samples at each training epoch, and $T^{\star}_{val}$ and $T^{\star}_{test}$ were fixed in a static manner for the overall spatial robustness validation and evaluation.

To measure the spatial robustness accuracy of a retrained model, we created a series of spatial test datasets with different levels of perturbation strengths. We specified the same five perturbation subranges for all translation operations in the GenRoust tool. These five perturbation subranges were $[-\omega_i\tau, -\omega_{i-1}\tau) \cup (\omega_{i-1}\tau, \omega_i\tau]$ for $\omega_i \in \langle 1/5, 2/5, 3/5, 4/5 \rangle$ (where $\omega_0 = 0$). For each ω_i, we configured the GenRobust tool to replace the original perturbation range with the corresponding perturbation subranges of all translation operations. We then ran the GenRobust tool with the original test dataset T_{test} as input to produce a spatial test dataset for that perturbation subrange, i.e., one spatial test dataset $T^{\star\omega}_{test}$ for one ω_i value.

(d) Retraining datasets in peer techniques. Although all the configured techniques take the adversarial training dataset (T^{*}_{train} or $T^{\star}_{train}$) as input, we noted that SENSEI, RobOT, DeepGini, and AugMix set up their own variants of training datasets. For brevity, we captured the main difference here and invited readers to refer to their papers for the full details.

SENSEI [28] implemented a genetic algorithm as the search strategy in its fuzzing component to search for representative training samples. We have reviewed SENSEI in Section 2. SENSEI initialized 10 chromosomes for each training sample to start its search process (where the value 10 for its hyper-parameter of chromosome population size was the best result in the original SENSEI experiment [28]). Among all the perturbed samples of each sample, SENSEI selected the one with the largest loss value as the representative sample for this training sample and used the representative sample to train the model in that epoch.

RobOT [48] and DeepGini [25] in our experiment need their own metrics to evaluate the training samples in T^{*}_{train} and $T^{\star}_{train}$ based on the ongoing training process in each epoch. Both of them prioritized the training samples by their individual metrics and then selected 10% (the best experiment setting in their papers) of samples for training on that epoch. We note that across different epochs, the selected training samples may be different from the last epoch as the model state was changed during the training process.

AugMix [69] included seven kinds of augmentation for training samples in its modified training dataset. During the training process, it used the Jensen-Shannon divergence consistency loss [71] as a regularization in the loss function.

4.2.5 Accuracy evaluation

We measured the performance of the retrained $\mathcal{F}$ by each technique on each (clean and noisy/spatial) test dataset to evaluate its standard or robustness accuracy. In Tables 3 to 7, we refer to the accuracy

on T_{test} and the accuracy on each of T^*_{test} and $T^\star_{test}$ as the standard accuracy (**std**), the noisy robustness accuracy (**nRob**), and the spatial robustness accuracy (**sRob**), respectively. We also refer to the sum of the standard and robustness accuracy as the joint accuracy (**joint**). For individual levels of perturbation strength, we denote the accuracy on $T^{*\sigma}_{test}$ by **blur**σ for each $\sigma \in \Sigma$ and the accuracy on $T^{\star\omega}_{test}$ by **spat**ω for each $\omega \in \Omega$. We also refer to them as the robustness accuracy at blurσ and spatω, respectively.

The **standard accuracy** of a model on T_{test} is defined as the proportion of samples in T_{test} predicted as the ground truth of the sample by the model. Similarly, the **robustness accuracy** of a model on each robustness dataset is defined as the proportion of samples in the dataset predicted as the ground truth of the sample by the model. The **joint accuracy** is the sum of the standard accuracy on T_{test} and the robustness accuracy on the spatial test dataset $T^\star_{test}$. We note that the experiment adopted the measure of the robustness accuracy for spatial robustness commonly used in the literature [12, 84].

4.2.6 Procedures

We conducted six experiments to evaluate DeepPatch for answering the research questions.

(a) To Answer RQ1. As presented in the previous sections, a core design goal of DeepPatch is to retain the standard accuracy while improving robustness. To answer RQ1, we designed two experiments: Experiment 1 and Experiment 2. They compared the full version of DeepPatch (with γ-PU activating) to all the peer techniques on their ability to robustness improvement and standard accuracy retention at the same time.

Experiment 1

In this experiment, we compared the following techniques on the Gaussian Blur noise robustness. We measured the accuracy of the models produced by each technique on the clean test dataset T_{test} and each noisy test dataset ($T^{*\sigma}_{test}$ and T^*_{test}). In the sequel, we present the procedure of each technique.

PT We used the pretrained model $\mathcal{F}$ in each benchmark as the model under maintenance, which were downloaded and not pretrained by us.

DP-γ We input the pretrained model, blamed ratio, the pair of noisy training and validation datasets (T^*_{train} and T^*_{val}), and the validation dataset T_{val} to our DeepPatch tool (Alg. 1) to generate a patched model. The blamed ratio κ was set to 25 by following the setting of DeepCorrect.

DC We used the DeepCorrect tool to produce a patched model inserted with the correction units. The correction units were trained with T^*_{train} and T^*_{val}, using the same method to train the replacement filters of DP-γ. The percentage of corrected filters in their paper [14] was set to 25%, meaningfully the same as the blamed ratio, so we also denoted it by κ. DeepCorrect got its best results by setting $\kappa = 75$ in their experiments. We thus repeated the experiment on DeepCorrect with $\kappa = 75$.

DC+FT We created the pipeline DC+FT, which further finetuned the model produced by DeepCorrect with $\kappa = 75\%$ on the training dataset T_{train} and validating on T_{val} with FineTune to serve as a simple baseline for standard accuracy recovery built on top of DeepCorrect.

AP We ported the Apricot tool into our framework to produce the retrained model. We ran the Apricot tool with the noisy training dataset T^*_{train} and the noisy validation dataset T^*_{val} as the input in its original algorithm. Since Apricot included generating 20 submodels for each pretrained model and repeatedly adapting their weights followed by finetuning, which consumed huge time, we cut off the execution after 24 hours and used the generated model with the highest noisy validation accuracy as its resultant model.

DG We created an adversarial training script and inserted the test case selection method of DeepGini into the process of feeding training samples to the model.

RB Similar to DG, we also created an adversarial training script for RB and implemented the test case selection method of RobOT.

AM AugMix (AM) has its tool available, and they implemented the code to generate their own datasets for their adversarial training steps. We added the gaussian blurring operation as

one of the augmentations in the tools. We input the training and validation datasets (i.e., T_{train} and T_{val}) to produce a resultant model.

Experiment 2

In this experiment, we compare the following techniques by evaluating on the clean test dataset T_{test} and each spatial test dataset ($T^{\star\omega}_{test}$ and $T^{\star}_{test}$).

PT Like Experiment 1, we used the pretrained model $\mathcal{F}$ in each benchmark as the model under maintenance.

DP-γ We repeated the procedure of DP-γ in Experiment 1 but using the spatial training dataset $T^{\star}_{train}$ and the validation dataset $T^{\star}_{val}$ to replace T^{*}_{train} and T^{*}_{val}, respectively.

SS SENSEI provided its tool implemented on Tensorflow. Due to the difference in the underlying platform, we ported it into our framework. We ran the SENSEI tool to produce an improved model, input with the training dataset T_{train} and validation dataset T_{val} as the tool was already configured with its fuzzing component to perturb samples.

DG, RB We repeated the procedure of DG and RB in Experiment 1, respectively, but configured with the spatial training dataset $T^{\star}_{train}$ and spatial validation dataset $T^{\star}_{val}$ instead.

AM We ran the original tool of AugMix input with the training and validation datasets (i.e., T_{train} and T_{val}) to produce a resultant model.

(b) To Answer RQ2. To answer RQ2, we conducted an ablation study on DeepPatch. We created two variants from DP-γ.

(1) We removed the γ-PU from DP-γ to produce the variant DP-β. This DeepPatch variant represented DeepPatch without applying the standard recovery task. The comparison between DP-β and DP-γ also shows the effect of the separation boundary.

(2) We removed the decision indicator to disable the adaptation behavior of DP-γ. We referred to the latter variant as the fallback option of DeepPatch (DP-fb). It was a patched model fully patched by DeepPatch but without activating the γ-PU in the patched model. Thus, DP-fb had the same model architecture and weight parameters as the patched model produced by DeepPatch (full algorithm), including all added filters and added logic, but they were not activated in any forward pass by forcing the decision indicator (see Section 3.2.3) to FALSE in making all decisions.

We compared them to better demonstrate the effects of β-PU and γ-PU in the patched models.

Experiment 3

This experiment performed an ablation study on DP-γ and evaluated for the standard accuracy and both noisy robustness and spatial robustness perspectives.

DP-γ We adopted the patched models generated by DeepPatch in Experiment 1 and Experiment 2.

DP-β We configured the DeepPatch tool by ablating the standard recovery task. More specifically, lines 19 to 22 of Alg. 1 were excluded and not executed. We repeated the procedures for DeepPatch in Experiment 1 and Experiment 2 but used this modified DeepPatch tool instead of the original DeepPatch tool.

DP-fb Echoing Section 3.2.3, making the decision indicator $\mathfrak{d}$ in the γ-PU to False can fall back our patched model to the behavior of the pretrained model $\mathcal{F}$. For each patched model generated in the DP-γ section above, we also cloned a copy of the patched model and forced the decision indicator $\mathfrak{d}$ in the cloned copy to be always False (see line 39 of Listing 1 for illustration) to produce a model with the fallback option enabled.

(c) To Answer RQ3. Experiment 4 evaluated to what extent DeepPatch can be benefited from SENSEI or be adversely affected by the latter. It also demonstrated how to configure possible pipelines of the two techniques. Since SENSEI was expected to reduce the standard accuracy of a retrained model it

produced, our aim was to evaluate different pipelines of SENSEI and DeepPatch for the feasibility of retaining the standard accuracy of the pretrained models and reaching the level of robustness improvement of DeepPatch when DeepPatch was applied in a standalone manner.

We combined SENSEI and DeepPatch to create two pipelines to evaluate DeepPatch further.

The SS+DP pipeline explored whether an improved model by SENSEI could boost its standard accuracy by training it further with DeepPatch using the clean dataset. More specifically, the filters in the model produced by SENSEI were trained on the spatial dataset, and the replacement filters added by DeepPatch were trained on the clean dataset. As such, the replacement filters and the filters originated from the model produced by SENSEI aim for processing samples more similar to the spatial validation dataset and the clean validation dataset, respectively. This pipeline represented an alternate use case of DeepPatch, which used the ability of replacement filters and dynamic decisions on filter activations in patching units to boost the standard accuracy.

The DP+SS pipeline explored whether a patched model by DeepPatch could improve its robustness by retraining the replacement filters of the patched models by the training dataset dynamically generated by SENSEI. The fuzzing component of SENSEI can generate valuable test cases for robustness improvement. The pipeline represents a use case of DeepPatch to integrate it with an adversarial test data generation technique.

Experiment 4

This experiment studied the effect of combining DeepPatch and SENSEI in a pipeline and evaluated the standard accuracy and the spatial robustness accuracy of the generated models.

PT We adopted the pretrained models of each benchmark.

SS We adopted the models generated by SENSEI in Experiment 2.

SS+DP The SS+DP pipeline was to apply the DeepPatch tool with T_{train} and T_{val} and the model produced by SS as inputs. We repeated Experiment 2 for DeepPatch but used the model output by SS instead of the pretrained model.

DP+SS The DP+SS pipeline applies the SENSEI's fuzzing-based data augmentation technique in DeepPatch's Alg. 1. More specifically, it accepts the pretrained model $\mathcal{F}$, $\kappa = 25$, and the clean validation dataset and the spatial validation dataset as input Instead of applying the adversarial training dataset generated by the original Alg. 1, the pipeline applies the SENSEI tool in line 18, in which, at each retraining epoch, the SENSEI tool generates its adversarial training dataset and retrains the replacement filters already unfrozen at line 17 of Alg. 1 for adversarial training.

(d) To Answer RQ4. DeepPatch was deeply inspired by DeepCorrect. Both techniques included a filter assessment task. They needed to specify a priority list of filters and select the top κ% filters from the list. To answer RQ4, we studied two aspects of DeepPatch related to this design.

The first aspect was whether it was worth selecting a subset of the filters for robustness improvement. To make the study general, we used a random selection of filters so that the results could be applied to both DeepPatch and DeepCorrect as a baseline. Experiment 5 evaluated the impact of the key parameter of this aspect, which was the blame ratio.

DeepPatch and DeepCorrect used their own filter assessment tasks to prioritize filters before their remaining robustness improvement (and standard accuracy retention) activities. Suppose that such a prioritized list was divided into a consecutive sequence of sections of equal size. The common underlying assumption of their filter assessment tasks was that the filters appearing in the first section of their prioritized lists should be preferable to the filters in later sections for achieving high overall robustness improvement after retaining these filters. Thus, the second aspect was whether this assumption could be consistently observed across the benchmarks. Experiment 6 validated this assumption. It evaluated whether the filters with top priority *consistently* achieved higher robustness accuracy than the filters with bottom priority.

Experiment 5

This experiment evaluated whether selecting a small set of filters could be beneficial.

We first set up a list of possible values to serve as the blamed ratio. We chose $\kappa \in \{15, 25, 35, 45, 55, 65, 75, 85, 95, 100\}$. We also modified the DeepPatch algorithm (Alg. 1) as follows. We changed line 11 to randomly select $\kappa\%$ of the filters of *every* convolutional layer irrespective of the accuracy difference computed in line 9. We also deleted the lines from line 19 to line 22. We then implemented the changes into our DeepPatch tool. For each possible κ value in the above list, we repeated the procedure for DeepPatch in Experiment 1 except that we ran the modified DeepPatch tool instead of the original DeepPatch tool and used the κ value in question instead of the fixed value of 25%.

Experiment 6

This experiment evaluated the filter assessment task in DeepPatch against the underlying assumption of filters with top priority in the model under maintenance are preferably selected over other filters. We measured the differences d_1 and d_2 in each noisy test accuracy by subtracting the accuracy of two models produced by DP and the two produced by DC, respectively. The sign of d_1 and d_2 across all noisy test datasets measured the consistency of filter assessment.

$\overrightarrow{\text{DP-}\alpha}$ We ran Alg. 1 with lines 1–11 and $\kappa = 25$ on each pretrained model, which assessed filters using the noisy dataset T^*_{val}, to get the top $\kappa\%$ filters in the prioritized queue Q. We then made the kernels of these filters in the pretrained model trainable and all other weights frozen and applied adversarial training using each noisy training dataset and the validation dataset to produce a retrained model.

$\overleftarrow{\text{DP-}\alpha}$ We repeated the above experiment except using the bottom 25% filters in the prioritized queue Q instead of the top 25% filters of Q.

$\overrightarrow{\text{DC}}$ We got the prioritized queue Q' of filters using DeepCorrect's algorithm (Alg. 1 in [14]) on each pretrained model and the noisy validation dataset and selected the top 25% filters from the queue Q'. Like what we described in the section on $\overrightarrow{\text{DP-}\alpha}$, we made the kernels of these filters in the pretrained model trainable and all other weights frozen and applied the same adversarial training using each noisy training dataset and the validation dataset.

$\overleftarrow{\text{DC}}$ We repeated the experiment described in the section on $\overrightarrow{\text{DC}}$ except that we selected the bottom 25% filters from the queue Q' instead of the top 25%.

4.3 Results and Data Analysis

4.3.1 Standard Accuracy Retention with Robustness Improvement

In this section, we compare DeepPatch with peer techniques to study how well the techniques can retain the standard accuracy with a boost in robustness compared to the pretrained models. Tables 3 and 4 summarize the experimental results for Experiment 1 and Experiment 2, respectively. The results of the techniques that maintain the standard accuracy within 1% absolute difference to Pratrained and that achieve the robustness accuracy within 5% relative difference to the highest are highlighted.

Descriptive Statistics of Pretrained (PT). In Table 3, across all six models, when the perturbation level increases from blur0.5 to blur3.0, the trend of the pretrained models (PT) is that the robustness accuracy drops by half from blur0.5 to blur1.0 and then another half from blur1.0 to blur1.5. Afterward, the future drop is relatively moderate. It reconciles an intuition that when the perturbation of samples increases, a DL model is harder and harder to recognize the perturbed samples correctly. Similarly, in Table 4, across the perturbation levels from small to large (i.e., from spat$^1/_5$ to spat$^5/_5$), PT experiences a significant drop in spatial robustness accuracy. For instance, the relative losses, computed as $1 - \text{spat}^5/_5 \div \text{spat}^1/_5$, are 32.9%, 32.5%, 45.9%, 44.6%, 22.5%, and 31.2%.

For case ⑥ in Table 4, after conducting the experiment, we realize that the downloaded pretrained model [77] is already robust against the spatial noise and quite accurate because the pretrained model is originally transferred from the ImageNet weights. (Readers may know that the ImageNet model is

Table 3: Results (%) of standard accuracy and noisy robustness

Index	Tech	κ	#Params	std	blur0.5	blur1.0	blur1.5	blur2.0	blur2.5	blur3.0	nRob
	PT	/	0.466M	**93.53**	91.40	40.53	21.73	17.60	16.09	15.18	42.29
	DC	25%	0.543M	56.64	57.08	54.83	53.11	51.95	49.26	47.28	52.87
	DC	75%	1.138M	69.15	68.82	66.30	63.86	60.92	57.83	54.83	63.10
	DC+FT	75%	1.138M	78.71	76.80	47.87	32.92	26.24	21.59	18.43	43.22
①	AP	/	0.466M	**93.20**	89.24	29.36	17.83	16.02	15.19	14.75	39.37
	RB	/	0.466M	85.66	85.19	81.39	77.01	72.58	67.96	61.76	**75.93**
	DG	/	0.466M	86.01	85.05	80.81	75.47	69.86	63.96	57.63	**74.11**
	AM	/	0.466M	91.63	91.29	84.89	65.12	41.33	29.75	24.94	61.28
	DP-γ	25%	0.583M	**93.53**	91.40	61.46	76.40	73.73	68.32	63.13	**74.90**
	PT	/	0.700M	**93.12**	90.80	37.82	22.38	18.77	17.18	16.26	42.33
	DC	25%	4.249M	70.71	69.80	67.34	64.83	61.11	57.38	53.69	63.55
	DC	75%	32.582M	61.24	60.98	59.42	57.11	54.95	52.15	48.56	56.34
	DC+FT	75%	32.582M	77.55	73.25	44.17	31.52	27.16	24.74	23.60	43.14
②	AP	/	0.700M	92.39	88.56	31.15	20.15	17.38	15.93	15.57	40.16
	RB	/	0.700M	85.89	85.77	81.69	77.03	72.21	67.27	62.24	**76.01**
	DG	/	0.700M	84.28	83.92	78.91	72.95	66.49	61.04	55.51	71.87
	AM	/	0.700M	91.46	91.10	84.65	65.78	44.05	30.94	24.40	61.77
	DP-γ	25%	0.859M	**93.05**	90.24	73.61	68.95	69.34	65.69	61.85	**76.03**
	PT	/	9.987M	**74.63**	69.64	24.14	10.13	5.36	3.51	2.63	27.14
	DC	25%	11.518M	35.07	34.00	32.14	30.83	29.40	27.10	23.82	30.34
	DC	75%	23.732M	42.28	42.06	39.99	37.60	35.24	32.64	29.45	37.04
	DC+FT	75%	23.732M	54.79	52.35	30.64	17.91	11.25	7.06	4.66	25.52
③	AP	/	9.987M	**73.84**	68.36	19.74	8.03	4.45	3.02	2.33	25.68
	RB	/	9.987M	66.57	65.03	55.76	47.13	40.88	34.53	28.08	48.28
	DG	/	9.987M	64.28	63.02	54.01	46.41	39.76	33.49	27.19	46.88
	AM	/	9.987M	72.05	71.62	60.98	40.94	22.30	12.06	7.11	41.01
	DP-γ	25%	12.338M	**74.63**	69.64	43.60	55.36	50.08	44.20	38.37	**53.69**
	PT	/	1.356M	**72.63**	67.20	23.58	11.11	6.87	4.82	3.61	27.11
	DC	25%	3.603M	36.26	36.14	34.25	32.88	31.08	29.05	26.83	32.35
	DC	75%	21.540M	33.70	33.48	32.14	31.44	31.11	29.85	27.96	31.38
	DC+FT	75%	21.540M	42.96	41.34	33.77	29.45	26.16	23.20	20.23	31.02
④	AP	/	1.356M	**71.73**	66.08	19.81	9.74	5.98	4.28	3.31	25.85
	RB	/	1.356M	62.86	61.15	55.16	50.56	45.25	39.70	34.35	49.86
	DG	/	1.356M	59.69	58.30	51.99	45.94	40.25	34.54	28.82	45.65
	AM	/	1.356M	**72.20**	72.01	63.24	47.77	28.63	16.96	10.55	44.49
	DP-γ	25%	1.660M	**72.27**	67.34	53.92	54.33	52.19	47.05	41.49	**56.00**
	PT	/	1.13M	**77.58**	73.66	50.46	39.33	34.20	31.35	29.83	48.06
	DC	25%	1.34M	51.15	51.13	48.51	47.67	46.87	46.00	45.08	48.06
	DC	75%	3.03M	53.92	53.23	50.81	49.58	48.56	47.90	47.20	50.17
	DC+FT	75%	3.03M	56.15	54.53	50.82	48.65	46.20	44.46	42.72	49.07
⑤	AP	/	1.13M	**76.67**	74.02	50.96	39.41	34.07	31.47	29.82	48.20
	RB	/	1.13M	70.45	72.81	69.08	65.80	62.96	60.22	57.31	**63.95**
	DG	/	1.13M	69.77	69.40	65.30	61.08	58.17	55.88	53.46	61.86
	AM	/	1.13M	**76.67**	75.91	69.45	61.55	54.01	47.48	43.01	61.15
	DP-γ	25%	1.41M	**76.76**	73.48	68.13	64.49	61.46	58.88	55.98	**65.50**
	PT	/	11.271M	**72.72**	65.80	35.54	18.48	10.55	6.64	4.71	30.63
	DC	25%	13.454M	34.79	33.91	32.99	31.91	30.32	28.77	26.24	31.27
	DC	75%	30.877M	36.14	36.05	35.34	34.56	32.88	31.52	29.03	33.64
	DC+FT	75%	30.877M	37.10	36.84	35.82	35.15	33.30	31.31	29.37	34.12
⑥	AP	/	11.271M	**72.60**	64.36	30.15	13.66	7.24	4.66	3.54	28.03
	RB	/	11.271M	59.26	58.34	53.85	49.76	45.42	40.42	36.16	49.03
	DG	/	11.271M	60.48	59.51	55.14	50.03	44.89	40.41	35.72	49.45
	AM	/	11.271M	68.71	68.24	56.49	39.08	25.35	17.30	12.02	41.02
	DP-γ	25%	14.061M	**72.72**	65.87	60.60	55.54	50.21	44.53	38.99	**55.49**

* #Params is the number of parameters in the model, std = standard accuracy, nRob = noisy robustness accuracy, and blur$X$ = accuracy of Gaussian noise with σ.

trained on tens of millions of training samples with spatial data augmentation, and the TinyImageNet model comes with a fraction of the dataset used in producing the ImagetNet model.)

Overall Results of DeepPatch. From Table 3 and Table 4, overall speaking, DeepPatch equipped with the γ-PU outperforms the peer techniques significantly, and the peer techniques either lose the standard accuracy greatly or can not achieve large robustness improvement.

DeepPatch attains almost the same standard accuracy as Pretrained. The margins of the absolute difference between Pretrained and DeepPatch on the noisy robustness task (Table 3) and the spatial

robustness task (Table 4) are 0.00–0.82 and 0.00–0.51, respectively, which are small.

From the noisy robustness improvement results (the nRob column) of Table 2, DeepPatch boosts the robustness over Pretrained by a large extent, ranging from 36.28% to 106.56%. Moreover, DeepPatch achieves the largest improvements among the peer techniques in all cases except for case ① with a very small margin of 1.03% to the highest.

From Table 4, in cases ① to ⑤, in terms of the overall spatial robustness improvement (the sRob column), DeepPatch also achieves significant improvements over Pretrained from 67.52% to 130.37%. The spatial robustness improvements made by DeepPatch are competitive with the peer techniques in the experiment.

The last column of Table 4 further shows the joint accuracy that illustrates the overall effects of standard accuracy retention and overall robustness improvement. Recall that the joint accuracy is the sum of the standard accuracy (the std column) and the spatial robustness accuracy (the sRob column). Compared with Pretrained, DeepPatch improves joint accuracy in all cases. The main reason is that DeepPatch retains the standard accuracy of Pretrained, and any improvement in robustness contributes to the difference in joint accuracy between DeepPatch and Pretrained.

In case ⑥, attributed to the TinyImage model originally being resilient to the spatial noise, DeepPatch can only improve this pretrained model to a small extent using the perturbation strengths specified by the SENSEI tool. In the section on threats to validity, we further discuss this issue and report the finding on DeepPatch by using training samples with spatial noise of greater perturbation strengths. In brief, DeepPatch can improve the robustness by 5.99%.

Like Pretrained, DeepPatch also experiences a drop when the perturbation increases. However, we observe that its drop between two consecutive perturbation levels is significantly more gentle than Pretrained in both Table 3 and Table 4. We have investigated the underlying reasons. In Table 3, we find that the models produced by DeepPatch used the original filters instead of the replacement filters to process almost all samples in the blur0.5 dataset. The ratio of samples processed by the replacement filters rapidly increases from blur0.5 to blur3.0. In Table 4, DeepPatch has a similar trend in that it tends to use original filters to process samples in the std dataset, and the ratio of samples processed by replacement filters generally increases from spat$^1/_5$ to spat$^5/_5$.

The number of parameters can be an indicator of the efficiency of the model inference. The fourth column (the #Params column) in Table 3 shows the number of parameters of the model produced by each technique on the noisy robustness improvement task. For the spatial robustness improvement task, the number of parameters of the patched models produced by DeepPatch is the same as those listed in Table 3, as the same blamed ratio is used in patching to the same model architecture. The other techniques (excluding DeepPatch and DeepCorrect) generate models with the same size as the pretrained models for both tasks. To avoid overloading readers with repetitive information, we do not show the #Param column in Table 4.

In Table 3, the patched models produced by DeepPatch contain slightly more parameters (in ratio) compared to the pretrained models. The sizes of these models are significantly smaller than the models produced by DeepCorrect. Other techniques do not alter the model architecture. The model sizes of their generated models are the same as the pretrained models.

The following sections present the individual comparison between DeepPatch and each peer technique. To better demonstrate the significance of DeepPatch, we collected all accuracy values in columns from blur0.5 to blur3.0 and from spat$^1/_5$ to spat$^1/_5$, if appropriate, for each technique to form a list, and conducted the Wilcoxon Signed Rank Test [85] (paired test for short) on a pair of lists with Bonferroni correction at the 5% significance level.

Comparison with DeepCorrect: Table 3 shows the results of Experiment 1 for DeepCorrect (DC) with $\kappa = 25$, DeepCorrect with $\kappa = 75$, and DeepCorrect followed by FT (DC+FT).

Across all models from top to bottom in Table 3, in standard accuracy, DeepPatch is significantly higher than both variants of DeepCorrect by at least 24.38%, 22.34%, 32.35%, 36.01%, 22.84%, and 36.58%, respectively. The results show that DeepCorrect's strategy of filter repair to process adversarial examples significantly compromises the generalization of the models to process clean samples. As expected, having a stage of standard accuracy recovery, DC+FT can improve the standard accuracy on top of DeepCorrect. However, this pipeline still has a clear and large gap (with margins of 14.48%, 15.5%, 19.84%, 29.31%, 20.61%, and 35.62%) to reach the standard accuracy level attained by DeepPatch across the six models. The difference in the effects of DeepPatch over DeepCorrect on retaining the standard accuracy is large. We believe that the relatively large loss in standard accuracy makes the models produced by DC challenging to replace the pretrained models for use.

Columns 6 to 12 of Table 3 show the noisy robustness accuracy at each perturbation level and

the overall noisy robustness accuracy. Across the robustness accuracy on all perturbation levels (i.e., blur0.5 to blur3.0), DeepPatch outperforms each variant of DeepCorrect (all three variants) in all 36 cases (100%). Moreover, at the same value of κ (i.e., 25), on all models except case ②, the robustness accuracy of DeepPatch at blur3.0 is higher than that of DeepCorrect at blur0.5. (Note that the noise added to the samples at blur3.0 is six folds of blur0.5.). In the remaining case (i.e., case ②), the robustness accuracy of DeepPatch at blur3.0 is 11.39% (= 1 - 61.85/69.80) lower than that of DeepCorrect at blur0.5, but the standard accuracy of DeepPatch is 31.59% (= 93.05/70.71 - 1) higher. The result shows a significantly higher effectiveness of DeepPatch in accurately inferring samples at different perturbation strengths than DeepCorrect.

From the table, the general trend is that DeepCorrect with $\kappa = 75$ outperforms DeepCorrect with $\kappa = 25$, which is consistent with the result presented in the original paper of DeepCorrect [14]. Across the blur0.5 to blur3.0 datasets on the six models, there are 36 cases in total. DeepCorrect with $\kappa = 75$ outperforms DeepCorrect with $\kappa = 25$ in 27 cases (75%). By conducting the paired test between DeepPatch and DeepCorrect with $\kappa = 25$ and between DeepPatch and DeepCorrect with $\kappa = 75$, both tests show that the respective two results are significantly different at the 5% significance level.

DeepCorrect with $\kappa = 75$ inserts two folds more correction units than DeepCorrect with $\kappa = 25$ to the same pretrained model in each case. Yet, from Table 3 and on the datasets from blur0.5 to blur3.0, adding 300% correction units brings small effects in making the generated model consistently infer noisy samples significantly more accurately. Observe that, at a relatively large perturbation level (blur2.0 to blur3.0), the drop in robustness accuracy of DeepCorrect with $\kappa = 75$ is smaller than that with $\kappa = 25$. It shows the merit of using more correction units in the models generated by DeepCorrect to process noisy examples with higher perturbation strengths. However, as indicated by the last column of Table 3, DeepCorrect with $\kappa = 75$ is only marginally more effective than DeepCorrect with $\kappa = 25$ in improving the overall robustness accuracy.

We have studied the difference between the models generated by DeepCorrect and DeepPatch and found that the feature map concatenation method of DeepCorrect results in channel disorder with respect to the original order of the channels in the pretrained models, which contributes to the difference in results. (Note that DL models tend to have strong connections between weights and neurons in the corresponding positions established through a training process.) Suppose a convolutional layer f contains two filters $\langle g_1, g_2 \rangle$, and filter g_1 is blamed for the robustness drop. DeeCorrect constructs $\hat{G}_{\oplus} = \{g_1^{'}\}$ which is a newly added filter appended with a correction unit, and the other filter not to be corrected is kept in $G_{\odot} = \{g_2\}$. Thus, DeepCorrect has to modify the output of the layer f (even if the original filters work better for some cases), which becomes the concatenation of $G_{\odot}(x)$ and $\hat{G}_{\oplus}(x)$, i.e., $f^{'} = [G_{\odot}(x)\ \hat{G}_{\oplus}(x)] = \langle g_2(x), g_1^{'}(x) \rangle$. DeepPatch maintains the existing order of filters with the enforcement of position alignment in its algorithm. It constructs a set of filters for patching $\tilde{G}_{\oplus} = \{g_{1,p=1}\}$ and a set of filters without any patching $G_{\odot} = \{g_{2,p=2}\}$. The output of β-PU is the sequence of channels, i.e., $P(x) = \langle g_{1,p=1}(x), g_{2,p=2}(x) \rangle$, in the same sorting order as the channels in the output of the original layer f.

On average, relative to the pretrained models, DeepCorrect with $\kappa = 25$ and 75 introduces 123.75% and 1104.16% new parameters, and DeepPatch introduces an average of 20.62% new parameters. DeepPatch can achieve both higher standard accuracy and higher robustness accuracy than DeepCorrect by introducing significantly fewer new parameters.

The overall comparison result between DeepPatch and DeepCorrect is that DeepPatch is more effective than DeepCorrect in the retention of standard accuracy and improvement in robustness accuracy with the generation of smaller models.

Comparison with SENSEI: We compare DeepPatch to SENSEI by comparing DP-γ to SS in Table 4.

SENSEI suffers from a severe loss in standard accuracy, where the drops in standard accuracy for cases ① to ⑥ are 32.44%, 64.89%, 27.14%, 45.59%, 27.88%, and 6.60%, respectively. On average, the loss incurred by SENSEI is 34.09%.

The result shows that the strategy of SENSEI to optimize a pretrained model with respect to the perturbed samples with the largest loss is less successful in making the resultant models retain the standard accuracy. In view of the current results on standard accuracy retention, the resultant models produced by SENSEI may not be practical to replace the pretrained models to process the original kinds of samples predicted well by the pretrained models.

In terms of the overall spatial robustness improvement (the sRob column), like DeepPatch, SENSEI outperforms Pretrained to a large extent in cases ① to ⑤ and can only improve the pretrained model in case ⑥ to a small extent.

Table 4: Results (%) of standard accuracy and spatial robustness

Index	Tech	std	spat$^{1}/_{5}$	spat$^{2}/_{5}$	spat$^{3}/_{5}$	spat$^{4}/_{5}$	spat$^{5}/_{5}$	sRob	joint
	PT	**93.53**	47.94	43.58	38.55	34.95	32.15	39.81	133.34
	SS	61.09	69.96	68.74	68.14	66.62	64.03	66.32	127.41
	AP	**93.13**	47.25	42.68	37.62	34.19	31.53	38.91	132.04
①	RB	62.64	73.46	72.44	71.17	69.52	67.46	70.80	133.44
	DG	44.25	70.36	69.30	67.44	65.08	62.70	66.62	110.87
	AM	91.12	51.92	48.11	44.67	40.49	36.90	43.93	135.05
	DP-γ	**93.53**	72.08	70.27	67.85	65.79	63.38	66.69	**160.22**
	PT	**93.12**	48.06	43.84	39.65	35.12	32.43	39.22	132.34
	SS	28.23	68.58	69.21	67.54	66.02	63.31	66.38	94.61
	AP	91.77	45.95	41.60	37.44	33.43	30.92	37.80	129.57
②	RB	68.04	72.46	71.84	70.19	68.31	65.94	69.51	137.55
	DG	55.36	70.89	69.70	68.30	66.14	63.94	67.82	123.18
	AM	90.49	50.61	47.57	44.09	39.97	36.67	43.52	134.01
	DP-γ	**93.12**	72.46	71.50	69.95	67.91	64.96	69.08	**162.20**
	PT	**74.63**	26.83	22.70	18.67	15.94	14.50	19.82	94.45
	SS	47.49	52.66	51.98	51.37	49.77	48.11	50.63	98.12
	AP	**73.74**	25.29	21.11	17.11	14.74	13.43	18.48	92.22
③	RB	27.92	45.79	44.75	42.76	40.16	37.24	42.03	69.95
	DG	26.86	42.81	40.87	38.65	35.49	33.47	38.71	65.57
	AM	67.93	29.51	27.00	23.01	19.78	16.99	23.22	91.15
	DP-γ	**74.63**	50.62	49.00	47.79	45.58	42.97	45.66	**120.29**
	PT	**72.63**	25.18	22.05	18.69	15.70	13.94	18.84	91.47
	SS	27.04	48.77	49.29	49.35	47.69	45.60	47.30	74.34
	AP	**71.94**	24.66	21.52	18.15	15.29	13.54	18.50	90.44
④	RB	11.30	44.93	43.26	41.02	39.19	37.37	41.28	52.58
	DG	11.06	41.11	39.73	37.81	35.79	34.00	37.55	48.61
	AM	70.08	27.71	26.24	22.99	20.09	17.72	22.78	92.86
	DP-γ	**72.63**	46.29	45.56	44.00	42.35	40.16	43.10	**115.73**
	PT	**77.58**	36.48	35.27	33.66	30.9	28.28	32.58	110.16
	SS	49.70	59.97	59.78	58.58	57.07	54.81	57.58	107.28
	AP	76.35	35.56	34.78	32.72	29.87	27.3	31.58	107.93
⑤	RB	47.22	66.61	65.95	64.66	62.32	59.12	64.02	111.24
	DG	34.52	53.35	53.22	50.9	48.25	44.46	50.30	84.82
	AM	**77.58**	36.48	35.27	33.66	30.90	28.28	32.58	110.16
	DP-γ	**77.58**	61.55	60.59	58.62	56.12	52.62	57.88	**135.46**
	PT	**72.72**	70.33	64.33	58.27	52.89	48.42	61.46	**134.18**
	SS	66.12	65.29	64.10	62.86	61.46	59.72	63.39	**129.51**
	AP	**72.72**	70.33	64.33	58.27	52.89	48.42	61.46	**134.18**
⑥	RB	**72.72**	70.33	64.33	58.27	52.89	48.42	61.46	**134.18**
	DG	**72.72**	70.33	64.33	58.27	52.89	48.42	61.46	**134.18**
	AM	**72.72**	70.33	64.33	58.27	52.89	48.42	61.46	**134.18**
	DP-γ	**72.21**	68.78	64.28	58.76	53.66	48.70	62.59	**134.80**

std = standard accuracy, nRob = noisy robustness accuracy, and blur$X$ = the accuracy of Gaussian noise with $\sigma = X$.

If we look at the robustness improvement alone, the improvement made by DeepPatch is smaller than that of SENSEI in five out of the six cases. However, this counting method does not tell the whole story. In view of the very low standard accuracy of these models produced by SENSEI, we cannot conclude that SENSEI outperforms DeepPatch in robustness improvement.

SENSEI generates samples with the largest losses among the respective candidates in each training epoch and trains the current model state on these samples. We have studied these training samples. We find that more than 90% of these training samples belong to the spat$^{5}/_{5}$ dataset. This also explains why SENSEI outperforms Pretrained by more than 10% on the spat$^{5}/_{5}$ dataset in case ⑥.

Both DeepPatch and SENSEI produced significantly more moderate relative losses from spat$^{1}/_{5}$ to spat$^{5}/_{5}$ than Pretrained, where the relative losses for DeepPatch are 12.0%, 10.35%, 15.1%, 13.2%,

14.5%, and 29.2%, and these for SENSEI are 8.5%, 7.7%, 8.6%, 6.5%, 8.6%, and 8.5%, respectively. Moreover, in absolute terms, their respective improvements over Pretrained are large. Across the board, SENSEI generally produces a larger positive effect on robustness improvement at each perturbation level than DeepPatch. Since the models output by DeepPatch can retain the standard accuracy, the result shows that SENSEI leans the retained models toward (seriously) forgetting how to accurately infer clean samples to gain the ability of higher robustness improvement.

Taking the spatial dataset series spat$^1/_5$ to spat$^5/_5$ as a whole, the paired test shows that the two techniques have a significant difference in robustness improvement at the 5% significant level. However, we observe that the difference in robustness accuracy between DeepPatch and SENSEI are all within 6%, which is relatively small since the differences between each of them and Pretrained are 25% or more. We tend to believe that the tradeoff between standard accuracy and robustness improvement made by DeepPatch is significantly more viable than SENSEI.

DeepPatch outperforms SENSEI in joint accuracy by 25.75%, 71.44%, 22.59%, 55.67%, 26.27%, and 4.08% with an average of 34.30% in all six cases (① to ⑥), respectively. The difference is large. SENSEI only gets higher joint accuracy than Pretrained in one case (3.67% in ③). In the remaining five cases, SENSEI loses its competitiveness by 5.93%, 37.73%, 17.13%, 2.88%, and 4.67%, respectively.

Comparison with Apricot: We first note that in case ⑥ of Table 4, Apricot cannot produce any model with higher spatial validation accuracy than the pretrained model in each retraining epoch in Experiment 2. Thus, it restores the parameters of the constructing model to the original parameters of the pretrained model. We have repeated the experiment on Apricot a few times to confirm this observation. To better show the effects of a technique, we exclude this case in comparison.

In the aspect of standard accuracy retention, Apricot only suffers small losses from 0.03% to 0.91% for the noisy robustness improvement task and from 0.22% to 1.35% for the spatial robustness improvement task. Both DeepPatch and Apricot attain high standard accuracy compared with the pretrained models.

For the results of robustness accuracy on the blur0.5 to blur3.0 datasets and the spat$^1/_5$ to spat$^5/_5$ datasets, there are 61 robustness accuracy values. DeepPatch achieves larger robustness improvements than Apricot in 60 cases (98.36%). The paired test result shows that there is a significant difference in robustness accuracy between DeepPatch and Apricot at the 5% significant level.

In the aspect of overall robustness improvement (sRob and rRob columns), the robustness accuracy achieved by Apricot is not higher than that of the corresponding pretrained models except case ⑤ in Table 3 with a small margin of 0.14%. DeepPatch outperforms Apricot significantly in this aspect.

Although Apricot retains the standard accuracy of the pretrained models, the reduction in robustness accuracy compared to the pretrained models consistently observed on all robustness test datasets is not a merit. DeepPatch is significantly more effective than Apricot when both aspects are considered.

Comparison with RobOT and DeepGini: Similar to the note on Apricot above, in case ⑥ of Table 4, both RobOT and DeepGini cannot produce a model with higher spatial validation accuracy in each retraining epoch. So, their outputs are just the same as the pretrained models. Similar to the discussion on Apricot, we exclude the comparison case ⑥ when discussing the aspect of spatial robustness accuracy.

In Table 3, RobOT and DeepGini lose quite heavily the standard accuracy for the noisy robustness improvement task. From cases ① to ⑥, RobOT results in standard accuracy losses with 7.87%, 7.23%, 8.06%, 9.77%, 7.13%, and 13.46%, respectively. Similarly, DeepGini produces losses of 7.52%, 8.84%, 10.35%, 12.94%, 7.81%, and 12.24%, respectively. The two techniques incur more severe losses in standard accuracy on the spatial robustness improvement task (Table 4) than Pretrained. RobOT loses 30.89%, 25.08%, 46.71%, 61.33%, and 30.36%, respectively, in cases ① to ⑤. DeepGini performs even worse than RobOT, and the losses are 49.28%, 37.76%, 47.77%, 61.57%, and 43.06%, respectively. The two techniques cannot retain the standard accuracy. All these results are also significantly lower than the result of DeepPatch on standard accuracy.

In a closer look at the datasets for individual perturbation strengths, among the columns for blur0.5 to blur3.0 and spat$^1/_5$ to spat$^5/_5$ for the benchmarks ① to ⑤ in the two tables, DeepPatch achieves larger robustness improvements in 33 over 61 cases (54.09%) than RobOT, and in 57 over 61 cases (93.44%) than DeepGini. Across the board, DeepPatch is more effective than DeepGini at all perturbation strengths, and outperforms RobOT at more perturbation strengths. The paired test shows that there are significant differences between DeepPatch and DeepGini and between DeepPatch and RobOT at the 5% significance level.

For the overall robustness improvement, in the nRob column of Table 3 and in the sRob column of Table 4, RobOT consistently achieves higher robustness accuracy than DeepGini except that for case ⑥ in Table 3, DeepGini outperforms RobOT by 0.42% only. To reduce the repeated comparisons, we

report the comparison of DeepPatch with the better one of the two techniques. Over the 11 cases in two tables, DeepPatch achieves higher performance in 7 cases (63.63%). In those cases where RobOT or DeepGini performs better than DeepPatch, we also observe that the absolute accuracy differences in the nRob and sRob columns are significantly smaller than that in the std column. For instance, in case ① of Table 3, the standard accuracy of RobOT is 7.87% lower than that of DeepPatch, and the difference in noisy robustness accuracy is only 1.03%.

The overall effect of standard accuracy retention with robustness improvement can be revealed by the joint accuracy (shown in the joint column in Table 4). DeepPatch significantly outperforms both RobOT and DeepGini by at least 20.07%, 17.92%, 71.97%, 120.10%, and 121.77% in cases ① to ⑤, respectively.

Even though these two techniques achieve higher robustness accuracy than DeepPatch in some cases, such great losses in standard accuracy cannot be ignored in replacing the pretrained models to be used in practice.

Comparison with AugMix: Same to the reason in the note on Apricot, in cases ⑤ and ⑥ of Table 4, AugMix also can only output the pretrained model to serve as its improved model. We thus also do not count these two cases in the following discussion.

In Table 3 and Table 4, AugMix slightly reduces the standard accuracy of the pretrained models. AugMix suffers 0.43%–4.01% with an average of 1.92% loss and 0.25%–6.70% with an average 2.86% loss in standard accuracy, respectively. AugMix in both tables sometimes fails to achieve the goal of standard accuracy retention.

In robustness improvement on individual perturbation strengths, from the columns for blur0.5 to blur3.0 and spat$^{1}/_{5}$ to spat$^{5}/_{5}$ (11 columns in total), there are 56 entries for each technique. DeepPatch outperforms Augmix in 45 out of these 56 entries (80.35%). At a closer look, DeepPatch consistently achieves higher robustness accuracy than AugMix on the spatial robustness improvement task in cases ① to ④. In terms of the noisy robustness improvement task, AugMix tends to have larger improvements on relatively small perturbation strengths (blur0.5 and blur1.0) than DeepPatch but fails to perform well on larger perturbation strengths (from blur1.5 to blur3.0). As a result, the standard accuracy of AugMix is not affected much and only suffers a slight deduction. The paired test result shows that these DeepPatch and AugMix have a significant difference at the 5% significance level.

In Table 3, AugMix experiences a quick drop in the noisy robustness accuracy when the perturbation strengths increase. For instance, from blur0.5 to blur1.0, the noisy robustness accuracy achieved by AugMix is within 15% of the standard accuracy of the pretrained models. Then, the robustness accuracy drops to nearly half of the standard accuracy when the perturbation strengths grow stronger to blur1.5 and then blur2.0. The robustness accuracy drops even more on blur2.5 and then blur3.0. Compared to AugMix, DeepPatch has a more moderate and smoother trend on noisy robustness accuracy. Its robustness accuracy on the blur3.0 dataset is still competitive with that achieved by AugMix on the blur1.5 dataset.

Moreover, in terms of the overall noisy robustness accuracy (the nRob column in Table 3), DeepPatch outperforms AugMix by 22.23%, 22.89%, 30.92%, 25.87%, 7.11%, and 35.28%, respectively, which are significant. Similarly, from Table 4 on the overall spatial robustness accuracy (the sRob column), AugMix also has significantly poorer performances than DeepPatch for cases ① to ④. AugMix achieves only half of the spatial robustness accuracy of DeepPatch in these four cases.

Answering RQ1 (Standard Accuracy Retention with Robustness Improvement). DeepPatch significantly outperforms the peer techniques in retaining the standard accuracy of pretrained models with significantly higher joint accuracy. The relatively low standard accuracy of DeepCorrect, SENSEI, RobOT, and DeepGini posts a serious question mark on their eligibility as a replacement option for pretrained models. Apricot and AugMix retain high standard accuracy but the effect of robustness improvement is significantly worse than DeepPatch.

4.3.2 Albation Study on the Effects of DeepPatch Variants

Tables 5 and 6 summarize the standard and robustness accuracy achieved by the variants of DeepPatch in Experiment 2 and Experiment 3, respectively. Columns of Tables 5 and 6 can be interpreted similar to Tables 3 and 4, respectively.

DeepPatch with the fallback option (DP-fb) makes the patched models behave like the pretrained models. The results in all columns of both tables validate this behavior of DP-fb. We have also analyzed the output of DP-fb on each sample in all training, validation, and test datasets (for both clean samples and adversarial examples) and confirmed that it is the same as the output of the corresponding pretrained

Table 5: Results (%) of DeepPatch variants on standard accuracy and noisy robustness

Index	Tech	std	blur0.5	blur1.0	blur1.5	blur2.0	blur2.5	blur3.0	nRob	joint
	DP-β	90.00	89.85	85.30	79.34	73.73	68.32	63.13	77.52	167.52
①	DP-γ	93.53	91.40	61.46	76.40	73.73	68.32	63.13	74.90	168.43
	DP-fb	93.53	91.40	40.53	21.73	17.60	16.09	15.18	42.29	135.82
	DP-β	88.65	88.92	84.13	77.48	72.17	66.97	61.85	77.17	165.82
②	DP-γ	93.05	90.24	73.61	68.95	69.34	65.69	61.85	76.03	169.08
	DP-fb	93.12	90.80	37.82	22.38	18.77	17.18	16.26	42.33	135.45
	DP-β	68.13	67.81	61.93	55.36	50.08	44.20	38.37	55.13	123.26
③	DP-γ	74.63	69.64	43.60	55.36	50.08	44.20	38.37	53.69	128.32
	DP-fb	74.63	69.64	24.14	10.13	5.36	3.51	2.63	27.14	101.77
	DP-β	68.34	67.13	61.90	56.59	52.28	47.05	41.49	56.40	124.74
④	DP-γ	72.27	67.34	53.92	54.33	52.19	47.05	41.49	56.00	128.27
	DP-fb	72.63	67.20	23.58	11.11	6.87	4.82	3.61	27.11	99.74
	DP-β	73.29	72.88	68.13	64.49	61.46	58.88	55.98	65.01	138.30
⑤	DP-γ	76.76	73.48	68.13	64.49	61.46	58.88	55.98	65.50	142.26
	DP-fb	77.58	73.66	50.46	39.33	34.20	31.35	29.83	48.06	125.64
	DP-β	66.88	66.10	60.60	55.54	50.21	44.53	38.99	54.69	121.57
⑥	DP-γ	72.72	65.87	60.60	55.54	50.21	44.53	38.99	55.49	128.21
	DP-fb	72.72	65.80	35.54	18.48	10.55	6.64	4.71	30.63	103.35

std = standard accuracy, nRob = noisy robustness accuracy, and blur$X$ = accuracy of Gaussian noise with $\sigma = X$.

Table 6: Results (%) of DeepPatch variants on standard accuracy and spatial robustness

Index	Tech	std	spat$^1/_5$	spat$^2/_5$	spat$^3/_5$	spat$^4/_5$	spat$^5/_5$	sRob	joint
	DP-β	54.94	77.16	75.90	74.00	72.00	69.29	73.48	128.42
①	DP-γ	93.53	72.08	70.27	67.85	65.79	63.38	66.69	160.22
	DP-fb	93.53	47.94	43.58	38.55	34.95	32.15	39.81	133.34
	DP-β	53.84	72.84	71.85	70.30	68.29	65.37	69.45	123.29
②	DP-γ	93.12	72.46	71.50	69.95	67.91	64.96	69.08	162.20
	DP-fb	93.12	48.06	43.84	39.65	35.12	32.43	39.22	132.34
	DP-β	23.89	52.33	50.91	49.25	46.90	44.21	47.98	71.87
③	DP-γ	74.63	50.62	49.00	47.79	45.58	42.97	45.66	120.29
	DP-fb	74.63	26.83	22.70	18.67	15.94	14.50	19.82	94.45
	DP-β	13.58	47.39	46.90	45.33	43.61	41.53	44.30	57.88
④	DP-γ	72.63	46.29	45.56	44.00	42.35	40.16	43.10	115.73
	DP-fb	72.63	25.18	22.05	18.69	15.70	13.94	18.84	91.47
	DP-β	46.03	61.67	60.71	58.68	56.18	52.68	57.93	103.96
⑤	DP-γ	77.58	61.55	60.59	58.62	56.12	52.62	57.88	135.46
	DP-fb	77.58	36.48	35.27	33.66	30.90	28.28	32.58	110.16
	DP-β	67.38	66.78	65.72	64.69	63.22	61.12	64.89	132.27
⑥	DP-γ	72.21	68.78	64.28	58.76	53.66	48.70	62.59	134.80
	DP-fb	72.72	70.33	64.33	58.27	52.89	48.42	61.46	134.18

std = standard accuracy, sRob = spatial robustness accuracy, and spat$^x/_5$ = spatial robustness within $^x/_5$ spatial ranges.

model on the same sample. Also, since the accuracy of each of DP-β and DP-γ is not identical to the corresponding pretrained model, these two variants of DeepPatch do not have the above property of DP-fb.

In Table 5, when the perturbation is relatively large, DP-γ and DP-β are identical in noisy robustness accuracy. We have also verified that their outputs are the same. The results indicate that all samples are handed by the replacement filters in both DeepPatch variants. We observe that from cases ① to ⑥, the boundaries of completing using the replacement filters for prediction are between blur1.5 and blur2.0, between blur2.5 and blur3.0, between blur1.5 and blur2.0, between blur2.0 and blur2.5, between blur0.5 and blur1.0, and between blur0.5 and blur1.0, respectively. The result indicates that the separation boundary lies around the first sharp drop observed in the trend of the performance of the pretrained models. In Table 6, we observe that the boundary of start using the replacement filters for prediction

is between the clean level and the first perturbation level (std to spat1/5). Up to the perturbation level of spat5/5 (which is the maximum perturbation strength used in the original experiment in [28]), the robustness accuracy of DP-γ is still lower than that of DP-β. We have analyzed the data and confirm that the replacement filters are activated in some but not all cases for the samples in spat5/5 and the proportions are increasingly larger from spat1/5 to spat5/5. The result indicates that when the perturbation level of a sample is relatively large, DeepPatch tends to guide its patched models to use the replacement filters to process samples. When the perturbation level of a sample is smaller (e.g., from blur1.0 to blur2.0 in Table 5 and from perturbation close to *std* in Table 6), the robustness accuracy of DP-γ is between the robustness accuracy of DP-β and DP-fb. It indicates that replacement filters and their original filter counterparts are both involved in processing different samples falling within this range of perturbation levels.

Like SENSEI, DP-β succeeds in improving the robustness but cannot retain the standard accuracy (i.e., with some obvious loss in standard accuracy), which makes DP-β with obvious undesirable limitations to be used in practice on these models. DP-β improves the robustness of a pretrained model by training the replacement filters with adversarial examples and using these replacement filters for handling all examples.

Compared to SENSEI in Table 4, DP-β achieves higher robustness accuracy than SENSEI in cases ①, ②, and ⑤ but trades off more standard accuracy, and SENSEI outperforms DP-β in cases ③ and ④. Their overall effects on robustness improvement are close to each other.

Furthermore, by further patching the models produced by DP-β with γ-PU, the loss in standard accuracy introduced by β-PU is significantly diminished. Across the two tables, the losses in standard accuracy incurred by DP-β are 3.5%–59.1% with an average of 21.04%, and the introduction of γ-PU closes the gap (with a loss of 0%–0.82% only), achieving the retention of standard accuracy. Furthermore, DP-γ improves the joint accuracy of cases ① to ⑥ of DP-β on the spatial noise type with 31.80%, 38.91%, 48.42%, 57.85%, 30.30%, and 2.53%, respectively.

Answering RQ2 (Ablation Study on DeepPatch). DeepPatch with β-PU achieves higher robustness than DeepPatch with γ-PU, but its standard accuracy's weakness makes the resultant models unsuitable to replace the pretrained models in general. DeepPatch with γ-PU significantly reduces the drawback of β-PU on poor standard accuracy and recovers the standard accuracy back to the level achieved by the pretrained models. It also shows a significant robustness improvement on the latter on the spatial robustness task. DeepPatch with the fallback option behaves like the pretrained models in accuracy.

4.3.3 Improving DeepPatch with the data augmentation of SENSEI

Fig. 3 summarizes the result of Experiment 4 on the pipelines of SS+DP and DP+SS. We recall that the role of DeepPatch in SS+DP is to use the replacement filters to boost the standard accuracy of the model produced by SENSEI; whereas its role in DP+SS is to generate a model with β- and γ-PUs and let SENSEI retrain the replacement filters in the DeepPatch generated model to improve the robustness. (We could not have a similar pipeline for DeepPatch and DeepCorrect because their changed filter sets conflict.)

In Fig. 3, the x-axis is a series of techniques, which, from left to right, are Pretrained, SENSEI, SS+DP, and DP+SS. The y-axis is the joint accuracy (with both standard and spatial robustness accuracy shown).

The SS+DP pipeline generally improves over SENSEI in joint accuracy by a large margin. In particular, we observe that the standard accuracy of SS+DP is significantly higher. However, from the figure, across all cases, the standard accuracy of SS+DP is lower than Pretrained. They are all lower than DP-γ in Table 3 in standard and joint accuracy, making them slightly inferior choices compared to DP-γ. However, the difference is marginal. It shows that SS+DP can be a viable pipeline.

The DP+SS pipeline achieves the same level of standard accuracy as DeepPatch and obtains higher robustness and joint accuracy than all DeepPatch variants in Table 6. Moreover, in case ②, the standard accuracy of DP+SS is slightly higher than DP-γ. This pipeline also outperforms the SS+DP pipeline. Specifically, the standard accuracy of DP+SS for ① to ⑥ are 93.53%, 93.12%, 74.63%, 72.63%, 77.58%, and 72.21%, respectively, and the corresponding robustness accuracy values are 68.41%, 70.83%, 46.16%, 44.22%, 57.20%, and 63.01%, respectively. The difference in robustness accuracy between DP-γ in Table 6 and DP+SS also shows the effect of using the stronger adversarial examples produced by SENSEI to train replacement filters is positive. Although case ⑥ in Fig. 3 has a smaller improvement compared with the pretrained model under this experimental setting, the effect of applying such a pipeline is consistent

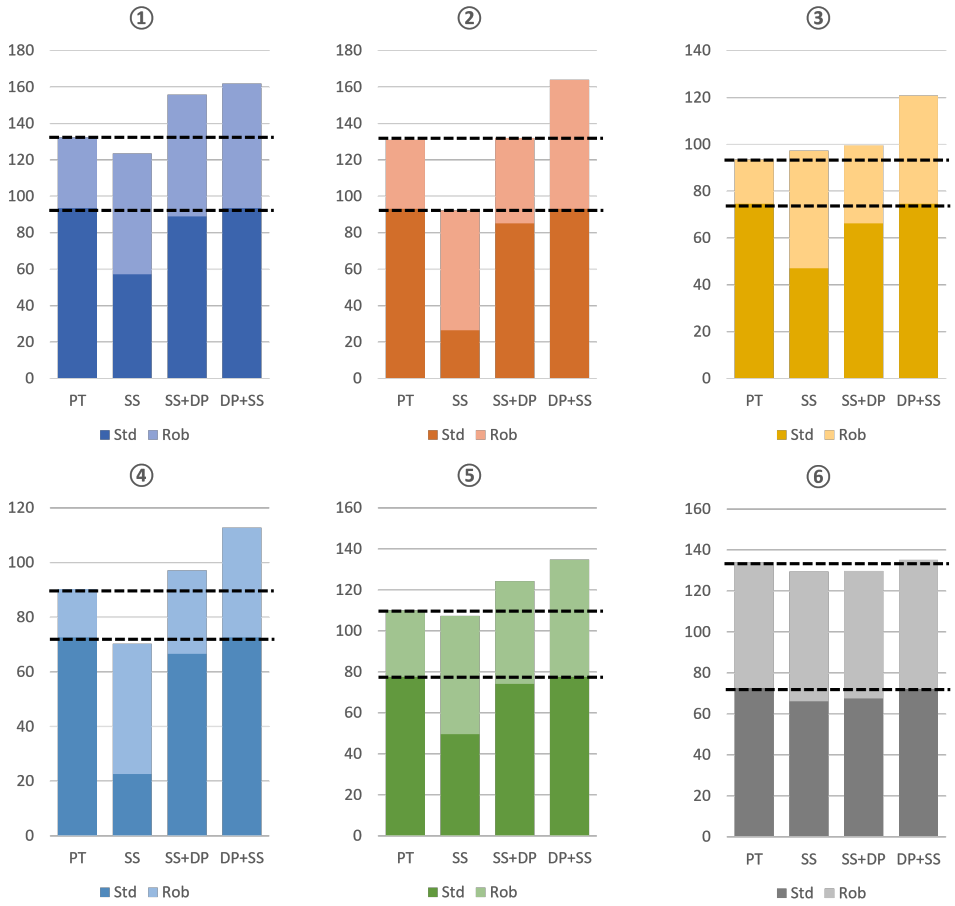


Figure 3: Comparison of joint standard and robustness performance of DeepPatch enhancement

with ① to ⑤. DP+SS demonstrates another effective pipeline.

Answering RQ3 (Effective Pipelines): DP+SS and SS+DP are effective pipelines, and DP+SS is superior to SS+DP in terms of both standard and robustness accuracy. Their performances are close to DP-γ. DP+SS is the most effective method among the experiment settings of SENSEI and DP pipelines and variants.

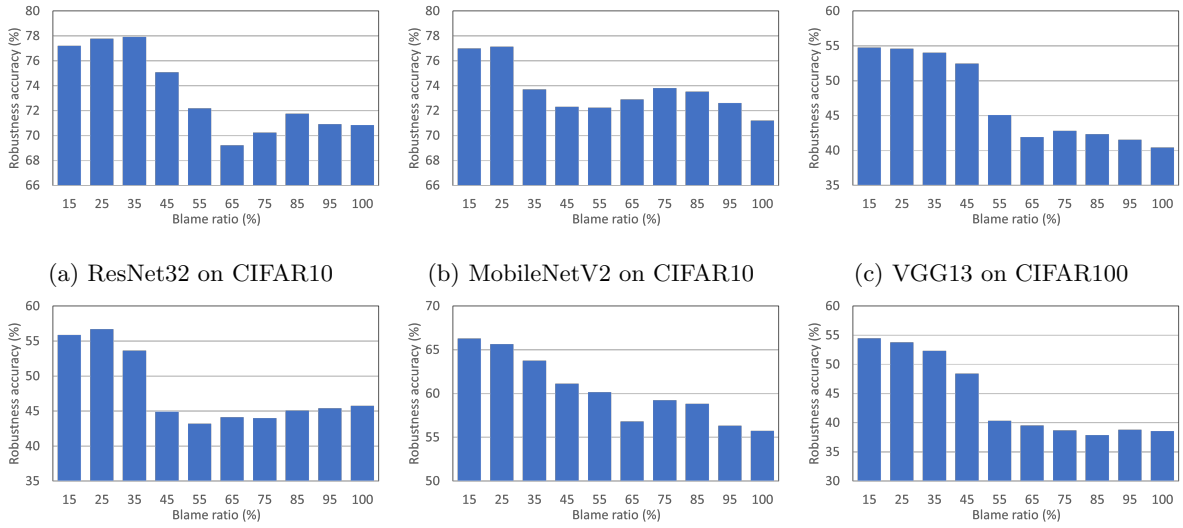
4.3.4 Blamed ratio

Fig. 4 shows the result of Experiment 5. One plot is for one case (①–⑥). The x-axis shows the κ varying from 15% to 100%, and the y-axis shows the corresponding robustness accuracy of the retrained models.

Across the six plots, as expected, the robustness accuracy of all retrained models was improved over the pretrained models (see Table 2). However, we observe that merely retraining more filters cannot achieve higher robustness accuracy than retraining fewer filters. Starting from $\kappa = 15$, as κ increases, the accuracy increases and then drops gradually. Generally, the highest performance happens on the relatively small.

4.3.5 Selective Filters Effects

Table 7 summarizes the accuracy achieved by DeepPatch with α-PU using top-25% filters ($\overrightarrow{\text{DP-}\alpha}$) and using the bottom-25% filters ($\overleftarrow{\text{DP-}\alpha}$) as well as DeepCorrect using the top-25% filters ($\overrightarrow{\text{DC}}$) and using the bottom-25% filters ($\overleftarrow{\text{DC}}$). The differences in test accuracy for the pair of $\overrightarrow{\text{DP-}\alpha}$ and $\overleftarrow{\text{DP-}\alpha}$ and for the pair of $\overrightarrow{\text{DC}}$ and $\overleftarrow{\text{DC}}$ are also shown. A positive difference means that the filter assessment task in the corresponding technique aligns with the underlying assumption. A positive and larger one between the two differences is highlighted in each column for each model-dataset combination. We examine the



(a) ResNet32 on CIFAR10 (b) MobileNetV2 on CIFAR10 (c) VGG13 on CIFAR100

(d) ShuffleNetV2 on CIFAR100 (e) Res-CNN on STL10 (f) ResNet18 on TinyImageNet

Figure 4: Robustness accuracy with different blamed ratios.

consistency among the signs of differences within each technique in all six cases (cases ① to ⑥) as a whole.

Among the 48 cases for the same technique, DP-α achieves 46 cases of positive difference (95.8%), but DC only has 21 cases (43.8%). The difference in percentage between DP-α and DC is 52%, indicating that DeepPatch's filter assessment task is more like to be consistent in ranking the filters for robustness improvement resulting in higher robustness improvement. The paired test result shows that these two techniques have a significant difference at the 5% significance level.

We next examine the rows showing the differences between the corresponding "top" and "bottom" rows. Among the 48 cases, DP-α achieves a larger difference in 46 cases than DC (95.8%) with an average difference of 4.25 in accuracy. For the remaining two cases, one case is that both DP-α and DC have no positive effect, and for the other case, DC only outperforms DP-α with a margin of 0.35. The average difference in ratio between DP-α and DC is 12.1x (4.25 ÷ 0.35).

In two cases (② and ④), there are larger differences in accuracy between $\overrightarrow{\text{DP-}\alpha}$ and $\overleftarrow{\text{DP-}\alpha}$. For instance, the differences in case ④ are more than 10% in both standard and robustness accuracy. On the other hand, there is almost no difference between $\overrightarrow{\text{DC}}$ and $\overleftarrow{\text{DC}}$.

Randomly picking filters subject to robustness improvement may accidentally pick less effective filters, such as these represented by $\overleftarrow{\text{DP-}\alpha}$ in cases ② and ④, which are undesirable. A made-up round of random selection may not necessarily result in better robustness improvement. On the other hand, $\overleftarrow{\text{DP-}\alpha}$ likely produces a more consistent and good result that aligns with the assumption of using the filter in the top section of a prioritized list generated by the DeepPatch filter assessment task is better. Comparing the accuracy in Table 7 with the pretrained models in Table 3, the drop in standard accuracy is still a problem, further justifying the need to use the full DeepPatch algorithm to patch models.

Answering RQ4 (Hyperparameter): The blamed ratio affects the robustness improvement to some extent, and in general, a small blamed ratio is preferable for high robustness improvement. DeepPatch can consistently find out effective filters for robustness improvement, outperforming DeepCorrect on selective filters from two angles of positive effects and larger margins.

4.4 Threats to Validity

The robustness improvement challenges come from adversarial training with large perturbed samples that harm the standard accuracy. Recently, there have been techniques to add adversarial training to standard training [38, 43]. In the experiment on CIFAR10 reported in the DAWNBench paper [43], on PreAct ResNet18, FGSM + DAWNBench [43] achieved 86.05% in standard accuracy, "Free" [38] 85.96%, "Free" + DAWNBench 78.38%, PGD-7 [35] 87.30%, and PGD-7 + DAWNBench 82.46%, all

Table 7: Results (%) of performance difference of retraining on noisy data with two sides selections

Index	Tech	Side	std	blur0.5	blur1.0	blur1.5	blur2.0	blur2.5	blur3.0	nRob
①	DC	top	89.01	88.86	83.71	77.35	71.41	65.59	60.26	76.59
		bottom	90.13	90.11	84.67	78.30	72.71	66.41	60.29	77.51
		difference	-1.12	-1.25	-0.96	-0.95	-1.30	-0.82	-0.03	-0.92
	DP-α	top	90.20	90.30	85.50	79.59	73.57	67.70	61.41	78.32
		bottom	89.40	89.27	84.29	78.24	72.12	66.97	61.50	77.29
		difference	**0.80**	**1.03**	**1.21**	**1.35**	**1.45**	**0.73**	-0.09	**0.93**
②	DC	top	88.26	88.01	83.96	78.66	72.84	68.39	62.73	77.55
		bottom	89.60	89.35	84.44	78.15	71.99	67.04	61.36	77.42
		difference	-1.34	-1.34	-0.48	0.51	0.85	1.35	1.37	0.13
	DP-α	top	89.40	89.20	84.64	78.61	73.27	68.17	63.01	78.04
		bottom	78.11	77.10	72.49	68.58	64.70	60.74	56.89	68.37
		difference	**11.29**	**12.10**	**12.15**	**10.03**	**8.57**	**7.43**	**6.12**	**9.67**
③	DC	top	68.32	67.84	60.98	54.19	47.93	42.54	36.77	54.08
		bottom	69.00	68.45	60.68	53.39	47.55	42.46	37.14	54.10
		difference	-0.68	-0.61	0.30	**0.80**	0.38	0.08	-0.37	-0.02
	DP-α	top	68.83	68.04	61.21	54.17	48.44	42.99	36.90	54.37
		bottom	68.32	67.91	60.70	53.72	47.41	41.90	36.72	53.81
		difference	**0.51**	**0.13**	**0.51**	0.45	**1.03**	**1.09**	**0.18**	**0.56**
④	DC	top	68.69	67.61	62.19	56.74	52.01	46.50	40.71	56.35
		bottom	68.30	67.94	62.08	56.56	51.32	46.62	40.21	56.18
		difference	0.39	-0.33	0.11	0.18	0.69	-0.12	0.50	0.17
	DP-α	top	67.35	67.10	61.86	57.12	51.70	46.18	40.66	55.60
		bottom	53.20	52.81	49.50	46.02	42.10	38.59	34.74	45.28
		difference	**14.15**	**14.29**	**12.36**	**11.10**	**9.60**	**7.59**	**5.92**	**10.32**
⑤	DC	top	72.56	72.00	68.22	65.08	62.51	60.10	57.40	65.41
		bottom	73.85	73.16	69.05	65.83	63.17	60.70	58.36	66.30
		difference	-1.29	-1.16	-0.83	-0.75	-0.66	-0.60	-0.96	-0.89
	DP-α	top	73.88	73.50	69.44	66.38	63.59	60.58	58.08	66.49
		bottom	73.27	72.15	68.38	65.37	62.98	60.68	57.78	65.80
		difference	**0.61**	**1.35**	**1.06**	**1.01**	**0.61**	-0.10	**0.30**	**0.69**
⑥	DC	top	65.81	65.00	59.88	55.31	49.84	44.61	38.23	54.09
		bottom	66.39	64.86	60.11	54.88	49.51	44.12	37.79	53.95
		difference	-0.58	0.14	-0.23	0.43	0.33	0.49	0.44	0.14
	DP-α	top	66.86	65.67	60.93	55.18	50.15	44.99	38.82	54.66
		bottom	64.67	63.57	59.27	54.09	48.98	44.47	37.85	52.84
		difference	**2.19**	**2.10**	**1.66**	**1.09**	**1.17**	**0.52**	**0.97**	**1.82**

std = standard accuracy, nRob = noisy robustness accuracy, and blur$X$ = the accuracy of Gaussian noise with $\sigma = X$.

with the target of robustness accuracy of 45%. Note that on the website[5], the standard accuracy of the pretrained PreAct ResNet18 on CIFAR10 is 95.11%, which is significantly higher than that of each model above, and the standard accuracy of ResNet18 (without the PreAct enhancement) is 93.02%. Our DP-γ on ResNet32 + CIFAR10 in our experiment retains the standard accuracy (93.53%) of this pretrained model with robustness accuracy of 74.90%. They reconcile the challenges tackled successfully by DeepPatch.

To decide the blamed ratio in DeepPatch for a given model in practical use, we formulate a methodology presented in Experiment 5. We have also experimented DeepPatch with $\kappa = 20$ and 30; the results are very close to the ones reported in the above experiment. Thus, we do not overload the readers with repetitive results for brevity.

We have trained more than two hundred models to evaluate DeepPatch from different aspects. The configurations include four widely used datasets, six pretrained models with significant model architecture differences, two kinds of noise used in the original work DeepCorrect and SENSEI, and eleven levels of perturbation strengths. The experiment includes six peer techniques, three variants of DeepCorrect, three variants of DeepPatch, a pipeline of DC+FT, and two pipelines of SS+DP and DP+SS. We believe our effort is comparable to the original experiments of included peer techniques and related works. Extending the work with more configurations may generalize the results.

The implementation of our framework and the downloaded tools of the peer techniques may contain

[5]https://github.com/kuangliu/pytorch-cifar

bugs. To make the experiment reliable, we tested it with the above configurations. We adopt the pretrained models and verify that they could replicate the accuracy published on the downloaded websites. All the crucial random processes have used a configurable and constant seed. The implementation code and the finalized maintained models are fully released to facilitate the replication of our experiment.

The experiment uses the full set of training samples to generate perturbed examples and retrain a model. We also observe that in the literature, to reduce the adverse effect on standard accuracy, in some experiments, researchers use a much smaller set of adversarial samples (e.g., 10% of generated data in RobOT [48] and 2% (=1000/50000) of training dataset in DeepRepair [19]). In general, restricting a model to retrain on a smaller dataset is quite a limitation, which is not in the spirit of deep learning to use more data to achieve better generalization. We have not conducted experiments using smaller retraining datasets.

We have only used the noise types and the noisy samples up to the noise levels presented in [14, 28]. Using other noise levels and other types of noise may produce different results. For example, we experimented to enlarge the perturbation strengths of each transformation two times in spatial datasets and then repeated Experiment 2 for case ⑥ on SENSEI and DeepPatch. The spatial robustness accuracy of the pretrained model dropped to 51.58%, and the standard accuracy was 72.72%. SENSEI suffered a 9.95% standard accuracy loss and achieved 58.52% spatial robustness accuracy, resulting in lowering the joint accuracy by 3.01%. While DeepPatch suffered 0.37% loss and achieved 57.94%, respectively, boosting the joint accuracy by 5.99%. We also repeated the DP+SS pipeline and evaluated that it could also improve the joint accuracy of 6.65%. We note that no other techniques in our experiment can outperform the pretrained model in case ⑥ on spatial robustness. The result validates the significant effect of DeepPatch on this benchmark.

DeepPatch requires a filter assessment task, which could be slow if the number of filters in the model under maintenance is large and become an obstacle in wide adoption in practice. In our experiment, assessing the VGG13 model (the model with the largest number of parameters in the experiment) only takes less than 10 minutes, and evaluating the other three models takes even short durations. We observe that this task can be completed in practical time. We also find that SENSEI requires a large computational resource during the training process to evaluate whether the current model state is already robust to each training sample. In the experiment, on average, SENSEI is 20 times slower than DeepPatch to output a model. DeepPatch also runs much faster than DeepCorrect due to DeepCorrect's slower filter prioritization strategy for processing each filter, which requires exchanging the feature maps among the different rounds of inferences on different samples. We limited the execution time of Apricot to 24 hours to generate a model. Using a large time budget may change the current results of Apricot reported in this paper.

4.5 Discussion

4.5.1 Use Cases

The idea of replacement filters to be dynamically activated and executed when needed is one of DeepPatch's main contributions. Originally, we aimed to explore the use of replacement filters to address the robustness improvement issue and the non-use of these filters to address the demand of retaining the original standard accuracy. In the experiment, the pipeline SS+DP demonstrates a new use case of DeepPatch to address the standard accuracy improvement issue, which also shows a promising result. SENSEI assesses several perturbed samples in each training epoch to select the one with the largest error to retrain the current model state. Intuitively, it produces stronger adversarial examples for retraining. The pipeline DP+SS shows that retraining the replacement filters with these stronger adversarial examples can outperform the same filters training with random adversarial examples (which is the result of DP-γ). Apart from SENSEI, there are other competing techniques to generate adversarial examples (e.g., [19]) or train with better mechanisms (e.g., [69]) in the literature. Integrating DeepPatch with these techniques may achieve larger robustness improvement. We leave the generalization of the experiment as future work.

We evaluated the effects of DeepPatch extensively in the experiment. Adding the γ-PU to a model with β-PUs will restore the original standard accuracy on demand, but it also lowers the robustness of the model when activated. We observe from the experiment that sometimes, the loss in robustness can be mild, but in many cases, the losses are quite noticeable. DeepPatch currently uses a one-master-many-slaves design style to organize all patching units in the same model. An alternate design may first divide the set of slave units into groups, and an independent master controls each group. We also leave the experiment on the alternate design as future work.

4.5.2 Filter assessment

The filter assessment task in DeepPatch nullifies the output of a filter via the α-PU, which is coarse. In the literature, coverage criteria related to how to track the influence of individual neurons, such as neuron path coverage [22], have been proposed. The filter assessment task can be more fine-grained by considering such coverage criteria to make the prioritization of filters more precise. However, these criteria can be quite computationally expensive. Integrating a lightweight version of such a criterion with DeepPatch warrants future research.

4.5.3 Finefuning the DeepPatch design

In the experiment, DeepPatch can be further configured to adjust the separation boundary toward the side of the smaller perturbation level to get a model with even higher robustness while retaining the standard accuracy. For instance, for a given model, developers may adjust the separation boundary by a certain percentage such as by evolving such an adjustment through an evolutionary algorithm to find a more promising model candidate. However, we have not formulated a systematic strategy to decide the step needed in such a boundary movement. We leave the investigation as future work.

Readers may consider an ensemble strategy of consisting of a classifier for making a binary decision on an input sample, the whole pretrained model, and the whole model produced by finetuning through typical adversarial training. The binary classifier decides which of the other two models to infer the input sample. Compared to DeepPatch, this strategy incurs redundant computations to produce the feature maps for the processing in the binary classifier and one of the other two models. The pretrained model and the retrained model as a whole double the memory consumption of the pretrained model in deployment. Experiment 6 used typical adversarial training, and using state-of-the-art adversarial training methods for finetuning may produce different results. It did not include any ensemble method for comparison. It is interesting to compare DeepPatch with ensemble methods and enhance the patching units of DeepPatch to improve ensembles. We leave the generalization of the experiment as future work.

We once explored and configured a classifier (inserted into the first convolutional layer in the patched model) to make the decision instead of using the current strategy by computing a CR score. We found that the robustness accuracy only has a marginal difference from our current experimental results reported in this paper but incurred higher latency. Therefore, we gave up this alternate design while formulating DeepPatch.

4.5.4 Generalization and Limitations beyond CNN models

Our experiments were only conducted on CNN models. To further evaluate the generalizability of DeepPatch to other possible kinds of DNN model architectures, we conducted a preliminary exploratory study[6] about the Transformer architecture [50] downlaoded from the popular open-source Pytorch platform [86]. Typically, a transformer model contains many multi-head attention layers, where a head is a core and the signature element in such a layer, equivalent to the role of a filter as a core and the signature element in CNN models. Vision Transformer is a well-known state-of-the-art model for image classification tasks [87].

We first executed the downloaded tool to generate a vision transformer model (i.e., ViT patch=4 Epoch@200 configuration in that platform [86]) on the CIFAR10 dataset (with a 49:1 split of the original training dataset into our training dataset and validation dataset) to produce a trained transformer model (i.e., the model under maintenance). We measured the standard accuracy of this model on the clean test dataset T_{test}. We also applied the noisy test dataset T^*_{test} to measure the noisy robustness accuracy. The standard accuracy and the noisy robustness accuracy of the trained transformer model were 79.36% and 49.16%, respectively.

DeepPatch processed each head [50] in each attention layer of the trained transformer model in the same way as it processed each filter in our above experiment on CNN models. We also made DeepPatch choose the first multi-head attention layer to insert a γ patching unit. Since each head also generated a feature map, we reused the code in the original DeepPatch tool to compute the corrective rate from the feature map generated by the original heads and the feature map generated by the replacement head counterparts in this attention layer.

Like our above experiment on CNN models, the modified DeepPatch tool took T^*_{train} and T^*_{val} as input and then patched the trained transformer model to create a replacement head for each head

[6]In software engineering, an exploratory study is a kind of empirical study to testify out a design able to work in a design context.

in each multi-head attention layer. It inserted both β- and γ- patching units into the patched model, followed by adversarial training on the noisy training dataset T^*_{train}. The modified DeepPatch tool finally produced the resultant patched model DP-γ.

We measured the standard accuracy and noisy robustness accuracy on the above-mentioned two test datasets. The standard accuracy and noisy robustness accuracy of DP-γ were 79.36% (same as the model under maintenance) and 65.84%, respectively. It means there was a robustness improvement of 16.68% (65.84% − 49.16%) while retaining the standard accuracy.

The exploratory study shows the flexibility of applying DeepPatch to Transformer-based models [50] with positive results. Moreover, DP-β may also be a viable choice for a patched model to improve the vision transformer model for noisy robustness improvement.

Although DeepPatch seems applicable to transformer models from this exploratory study, we also observe a limitation of DeepPatch. In some model architectures such as the Recurrent Neural Network (RNN) architecture [88], the current design of DeepPatch seems to require further refinement. DeepPatch assumes the first invocation of the "master unit" (the γ patching unit) to compute the corrective rate and passes the result to all other slaves (β patching units) in the same patched model. RNN models can be viewed unfolded during inference. Take the vanilla RNN architecture for discussion. The same hidden layer in an RNN model may be invoked multiple times. In principle, one β patching unit can be attached to each unfolded iteration of the same hidden layer, producing a series of β patching units. In the spirit of DeepPatch's replacement filter strategy, the replacement hidden states across different β patching units in the series will have likely been retrained to result in non-identical parameters. Thus, such a patched model will contain a set of parameter sets, one for each β patching unit in the series. As a comparison, in the RNN model, there is only one set of parameters for the same hidden layer. It becomes not obvious to us how to efficiently conduct adversarial training, e.g., the training script of the RNN model may not be directly usable for adversarial training to train these patching units. The use of the series of β patching units inserted into an RNN model essentially means converting the RNN model into a version without recursive invocations of the same hidden layer in every forward pass, defying the overall architectural design of RNN models. It may be a wise choice to design a variant of DeepPatch able to deal with the recursive invocations of the same hidden layer in a patched model while able to allow efficient training and inference and keeping the essence of RNN architecture design.

DeepPatch requires white-box access at the code level in order to perform the filter assessment task and the patching task. Eliminating such requirements may be interesting and may enlarge the adoption of DeepPatch in practice. We leave the generalization of DeepPatch on more model architectures and fewer limitations as future work.

4.5.5 Overheads of DeepPatch in the inference time

In traditional DNN models or even the models patched by each peer technique in our experiment, every unit in a model will be executed in every forward inference on every sample.

However, a novel design of DeepPatch is that in every forward inference on every sample, only either the original filters or the replacement filters (but not both) in each β patching unit will be executed to generate feature maps. The decision of choosing which set of filters to execute within each β patching unit is merely a simple condition of two Boolean variables on whether the decision indicator keeps a particular TRUE value, which its overhead can be ignored. (Readers may recall from Sections 3.2.2 and 3.2.3 about the setup and the use of the decision indicator in DeepPatch.) So, more filters within the set of β patching units in the patched model inserted by DeepPatch will only minimally increase the speed overhead and has no impact on the memory space occupied by their generated feature maps.

On the other hand, the main overhead of DeepPatch is in the computation taken in the first convolutional layer of the patched model. These overheads include the computation overhead of the Corrective Rate as the computation for the feature map generation for all the filters in the γ patching unit of the patched model. In the worst case, the γ patching unit doubles a feature map doubling the size of the feature map of the first convolutional layer of the model under maintenance. In the model architectures used in our experiment, only a small ratio of filters is from the first convolutional layer. Take the ResNet32 model in case ① for discussion. There are 1232 filters in the pretrained model and 308 filters are introduced in the blamed filters by DeepPatch, where only 4 filters come from the first convolutional layer. The memory overhead is almost exclusively due to the additional memory required to keep the feature maps generated by the replacement filters in the first convolutional layer only. The speed overhead in computing the CR metric requires comparing the two feature maps (generated by the replacement filters and their original counterparts). It essentially means inserting one more layer

for computation. The additional slowdown of the current implementation of the DeepPatch tool on the test datasets (by normalizing the speed of the pretrained models to 1) for cases ① to ⑥ are 1.009x, 1.067x, 1.287x, 1.090x, 1.278x, and 1.472x, respectively, with an average of 1.335x. As a comparison, the corresponding values of DeepCorrect are 1.359x, 2.000x, 1.069x, 1.909x, 1.798x, and 1.699x, respectively, with an average of 1.582x. The patched models produced by DeepPatch are more efficient than the patched models produced by DeepCorrect by 18.4% on average.

5 Related Work

This work is mainly related to the following works toward building highly accurate and robust deep learning models.

5.1 Deep Learning Testing

Many existing deep learning model debugging and repair works improve the model generalization through "buggy" components localization and fixing [14–16, 89–91]. MODE [16] localizes the "buggy" neurons via differentiation on two sets of correctly classified and misclassified samples, and then introduce more real-world data for finetuning. Apricot [15] generates a set of smaller models from the model under maintenance. The combined weights change direction is regarded as the correct direction to adjust the model's weights under maintenance. The weights are treated as the "buggy" components. Similarly, Xie et al. [91] build an inference model to diagnose the model under maintenance for the weight states and assign confidence scores for them. When localizing the brittle states, those weights are tuned for repair.

DeepPatch follows this line of research but focuses on the "buggy" filter components as the repair subjects. The difference is that DeepPatch performs the repair job on the "buggy" components only as incremental repairs. The above existing works on the other hand perform the repair at the model level, which also alters non-buggy components.

5.2 Input Prioritization

Another area of related techniques is prioritization techniques [25, 28, 48, 92–96]. DeepGini [25] prioritizes test inputs by computing the Gini index as the score for prioritization. The calculation of the Gini index is based on the predictive probabilities of classification tasks. Likely, Byun et al. [93] measure the model inference. It proposes three kinds of gauges relative to the input, confidence, surprise, and uncertainty, as the priority score. PRIMA [92] prioritizes an input with higher priority if the input causes more mutated model variants to predict incorrectly or more variants of the input cause the model under test to predict incorrectly. Shen et al. [95] select the samples near the classification boundary. Wang et al. [48] propose quantitative measurements with two stages. It firstly generates the variants of a sample for measurement and selects those high-quality samples for repairing empirically and greedily. The prioritized samples in the above techniques are then used to finetune the model.

DeepPatch is also equipped with a prioritization technique, but the subjects are neuron network components (filters) rather than data. DeepPatch assesses filters based on the impact on the model's robustness and selects the filters to be blamed for patching.

5.3 Deep Learning Model Maintenance

The primary goal of the above two lines of work is to evolve a DL model to achieve higher performance or mitigate current defects [14, 19, 97–100]. This goal is akin to software maintenance.

A kind of approach in this line mainly introduces new real-world data in maintenance. DeepFault [97] synthesizes inputs according to the gradients of the top-k suspicious neurons. DeepRepair [19] utilizes new data to perform a style-constraint augmentation on original data. Collecting data with specific styles of errors is non-trivial in the real-world environment, and the collected new data requires human labeling as the ground truth for the supervised learning tasks, which is labor expensive. In the experiment [19], DeepRepair adopted just one existing well-labeled dataset with such styles [101] for evaluation, which is the CIFAR10 dataset, and the noise data comes from the CIFAR10-C dataset. Compared with DeepRepiar on the Gaussian blur noise levels, DeepRepair achieves 75% robustness accuracy for three models trained with a perturbation strength less than 1.0, and DeepPatch also achieves the same level of robustness accuracy of 79.18% (taking the average of blur0.5 and blur1.0) but trained with

larger perturbation strengths up to 3.0. DeepPatch does not require new real-world data for model generalization to make it effective in robustness improvement, albeit using more data certainly helps.

Contrarily, some approaches augment the existing data and involve fuzzing techniques to search for a more optimized model state. Both SENSEI [28] and RobOT [48] fall in this category. RobOT generates sample variants along the direction of increasing the first-order loss on the gradient and then retrains the model with these variants. We have reviewed SENSEI in the previous section. Fuzzing is known to be computationally expensive in the training job, and optimizing toward many strong adversarial samples generally harms the standard accuracy. In the experiment, we have included two DeepPatch pipelines to explore the fuzzing effects. Currently, DeepPatch uses simple data augmentation without a heavy fuzzing component. Compared with RobOT, DeepPatch targets diverse and non-trivial perturbation strengths and retention of the standard accuracy. In the experiment of RobOT [48], the retrained models produced by RobOT still experience a loss in standard accuracy, even though RobOT uses only 10% of generated samples for retraining and only uses the samples with very small perturbation strengths by FGSM and PGD when targeting the retention of standard accuracy. DeepPatch revises the architecture program by inserting a unit to dynamically compute the CR metric value to invoke the original or replacement filters to address this issue.

Another kind of approach is to evolve the model with more parameters. DeepCorrect inserts correction units to evolve the model, but all the inputs have to go through correction units which may error process old inputs. Neural network search and evolution [99, 100] incrementally evolve the model architecture with retraining in each step. They generate and train numerous intermediate architectures, which are extremely computationally expensive. The evolution method of DeepPatch is more lightweight than these search-and-evolution-based methods.

5.4 CNN-based Software Engineering Applications

CNN models (e.g., [102]) are widely used in software engineering [103–107]. For instance, Zhao et al. [103] recognize workflow actions in programming screencasts by CNN models. Carlos et al. [108] use a VGG model to translate video recordings of Android apps to detect user behaviors and generate replayable actions. Li et al. [107] locate the buggy code by transferring the coverage matrices to the CNN model for recognition. DeepPatch has the potential to integrate with them to improve the effectiveness of other software engineering techniques.

6 Conclusion

In this paper, we have presented DeepPatch, a novel technique to patch convolutional neuron network models to retain the standard accuracy of the models under maintenance with significant boosts in robustness. DeepPatch consists of three components for the filter assessment, model patching, and standard accuracy recovery tasks, respectively, through the support of our α-, β-, and γ-patching units, respectively. Its patched models can dynamically activate and execute replacement filters or the original filter counterparts to infer input samples resembling more like the samples in the noisy validation dataset and those in the clean validation dataset, respectively. The patched models can also be fallen back to behave like the model under maintenance. Besides the final output, DeepPatch also generated an intermediately patched model inserted with only β patching units for possible use. Experiments have shown that DeepPatch successfully retains the original standard accuracy with a large extent of robustness improvement across benchmarks, which is not achievable by the peer techniques. DeepPatch is also experimented to be applied to the Visual Transformer model to explore its generalizability to different kinds of DNN architecture. The result of the exploratory study also shows that DeepPatch can retain the standard accuracy with a significant boost in noisy robustness.

References

[1] C. Chen, A. Seff, A. L. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of 2015 IEEE International Conference on Computer Vision, ICCV*. IEEE Computer Society, 2015, pp. 2722–2730.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of 2015 IEEE International Conference on Computer Vision, ICCV*. IEEE Computer Society, 2015, pp. 1026–1034.

[3] J. G. Richens, C. M. Lee, and S. Johri, "Improving the accuracy of medical diagnosis with causal machine learning," *Nature Communications*, vol. 1, pp. 24–35, 2019.

[4] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan 2016. [Online]. Available: https://doi.org/10.1038/nature16961

[5] N. Brown and T. Sandholm, "Superhuman ai for heads-up no-limit poker: Libratus beats top professionals," *Science*, vol. 359, no. 6374, pp. 418–424, 2018. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.aao1733

[6] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[7] M. Du, F. Yang, N. Zou, and X. Hu, "Fairness in deep learning: A computational perspective," *IEEE Intelligent Systems*, vol. 36, no. 4, pp. 25–34, 2021.

[8] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, pp. 61–67, 2019.

[9] S. Ma, Y. Liu, G. Tao, W. Lee, and X. Zhang, "NIC: detecting adversarial samples with neural network invariant checking," in *Proceedings of 26th Annual Network and Distributed System Security Symposium, NDSS*. The Internet Society, 2019.

[10] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 4901–4911.

[11] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," *Commun. ACM*, vol. 62, no. 11, p. 137–145, Oct. 2019.

[12] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 303–314.

[13] Z. Li, X. Ma, C. Xu, C. Cao, J. Xu, and J. Lü, "Boosting operational dnn testing efficiency through conditioning," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 499–509.

[14] T. S. Borkar and L. J. Karam, "Deepcorrect: Correcting dnn models against image distortions," *IEEE Transactions on Image Processing*, vol. 28, no. 12, pp. 6022–6034, 2019.

[15] H. Zhang and W. K. Chan, "Apricot: A weight-adaptation approach to fixing deep learning models," in *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering, ASE*. IEEE, 2019, pp. 376–387.

[16] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 175–186.

[17] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1135–1146.

[18] M. Sotoudeh and A. V. Thakur, "Provable repair of deep neural networks," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 588–603.

[19] B. Yu, H. Qi, Q. Guo, F. Juefei-Xu, X. Xie, L. Ma, and J. Zhao, "Deeprepair: Style-guided repairing for deep neural networks in the real-world operational environment," *IEEE Transactions on Reliability*, pp. 1–16, 2021.

[20] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 120–131.

[21] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 1039–1049.

[22] X. Xie, T. Li, J. Wang, L. Ma, Q. Guo, F. Juefei-Xu, and Y. Liu, "Npc: Neuron path coverage via characterizing decision logic of deep neural networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, apr 2022. [Online]. Available: https://doi.org/10.1145/3490489

[23] Z. Li, M. Pan, T. Zhang, and X. Li, "Testing dnn-based autonomous driving systems under critical environmental conditions," in *Proceedings of the 38th International Conference on Machine Learning, ICML*, ser. Proceedings of Machine Learning Research, vol. 139. PMLR, 2021, pp. 6471–6482.

[24] J. Chen, Z. Wu, Z. Wang, H. You, L. Zhang, and M. Yan, "Practical accuracy estimation for efficient deep neural network testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 30:1–30:35, 2020.

[25] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "Deepgini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 177–188.

[26] W. Shen, Y. Li, L. Chen, Y. Han, Y. Zhou, and B. Xu, "Multiple-boundary clustering and prioritization to promote neural network retraining," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 410–422.

[27] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *Proceedings of 43rd IEEE/ACM International Conference on Software Engineering, ICSE*. IEEE, 2021, pp. 397–409.

[28] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1147–1158.

[29] Z. Li, X. Ma, C. Xu, J. Xu, C. Cao, and J. Lü, "Operational calibration: Debugging confidence errors for dnns in the field," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 901–913.

[30] K. Muhammad, A. Ullah, J. Lloret, J. D. Ser, and V. H. C. de Albuquerque, "Deep learning for safe autonomous driving: Current challenges and future directions," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 7, pp. 4316–4336, 2021.

[31] C. Said. (2018) Video shows uber robot car in fatal accident did not try to avoid woman. [Online]. Available: https://www.sfgate.com/business/article/Uber-video-shows-robot-car-in-fatal-accident-did-12771938.php

[32] Jan 2019. [Online]. Available: https://www.tesla.com/en_HK/VehicleSafetyReport

[33] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *CoRR*, vol. abs/1710.08864, 2017. [Online]. Available: http://arxiv.org/abs/1710.08864

[34] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6572

[35] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[36] L. Engstrom, B. Tran, D. Tsipras, L. Schmidt, and A. Madry, "Exploring the landscape of spatial robustness," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 1802–1811. [Online]. Available: https://proceedings.mlr.press/v97/engstrom19a.html

[37] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.5068

[38] A. Shafahi, M. Najibi, A. Ghiasi, Z. Xu, J. Dickerson, C. Studer, L. S. Davis, G. Taylor, and T. Goldstein, *Adversarial Training for Free!* Red Hook, NY, USA: Curran Associates Inc., 2019.

[39] Y.-Y. Yang, C. Rashtchian, H. Zhang, R. R. Salakhutdinov, and K. Chaudhuri, "A closer look at accuracy vs. robustness," in *Proceedings of Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 8588–8601.

[40] A. Raghunathan*, S. M. Xie*, F. Yang, J. Duchi, and P. Liang, "Adversarial training can hurt generalization," in *ICML 2019 Workshop on Identifying and Understanding Deep Learning Phenomena*, 2019. [Online]. Available: https://openreview.net/forum?id=SyxM3J256E

[41] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[42] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)," 2009. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[43] E. Wong, L. Rice, and J. Z. Kolter, "Fast is better than free: Revisiting adversarial training," in *International Conference on Learning Representations*, 2020.

[44] H. Zhang, Y. Yu, J. Jiao, E. P. Xing, L. E. Ghaoui, and M. I. Jordan, "Theoretically principled trade-off between robustness and accuracy," in *International Conference on Machine Learning*, 2019.

[45] D. Su, H. Zhang, H. Chen, J. Yi, P.-Y. Chen, and Y. Gao, "Is robustness the cost of accuracy? – a comprehensive study on the robustness of 18 deep image classification models," in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[46] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness may be at odds with accuracy," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=SyxAb30cY7

[47] P. Tabacof and E. Valle, "Exploring the space of adversarial images," *CoRR*, vol. abs/1510.05328, 2015. [Online]. Available: http://arxiv.org/abs/1510.05328

[48] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "Robot: Robustness-oriented testing for deep learning systems," in *Proceedings of 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 300–311.

[49] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *CoRR*, vol. abs/1511.08458, 2015. [Online]. Available: http://arxiv.org/abs/1511.08458

[50] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[51] H. Wang, J. Xu, C. Xu, X. Ma, and J. Lu, "Dissector: Input validation for deep learning applications by crossing-layer dissection," in *Proceedings of 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020, pp. 727–738.

[52] Y. Yuan, W. Lu, B. Feng, and J. Weng, "Steganalysis with cnn using multi-channels filtered residuals," in *Proceedings of Cloud Computing and Security*, X. Sun, H.-C. Chao, X. You, and E. Bertino, Eds. Cham: Springer International Publishing, 2017, pp. 110–120.

[53] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," 2017.

[54] N. Khan and I. Stavness, "Pruning convolutional filters using batch bridgeout," *IEEE Access*, vol. 8, pp. 212 003–212 012, 2020.

[55] M. Lin, R. Ji, S. Li, Y. Wang, Y. Wu, F. Huang, and Q. Ye, "Network pruning using adaptive exemplar filters," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–10, 2021.

[56] E. Radiya-Dixit and X. Wang, "How fine can fine-tuning be? learning efficient language models," in *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]*, ser. Proceedings of Machine Learning Research, S. Chiappa and R. Calandra, Eds., vol. 108. PMLR, 2020, pp. 2435–2443.

[57] S. M. Xie, T. Ma, and P. Liang, "Composed fine-tuning: Freezing pre-trained denoising autoencoders for improved generalization," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 139. PMLR, 2021, pp. 11 424–11 435.

[58] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," 2013. [Online]. Available: https://arxiv.org/abs/1312.6199

[59] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," 2016.

[60] T. Borkar, F. Heide, and L. Karam, "Defending against universal attacks through selective feature regeneration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[61] A. Bianchi, M. R. Vendra, P. Protopapas, and M. Brambilla, "Improving image classification robustness through selective cnn-filters fine-tuning," 2019. [Online]. Available: https://arxiv.org/abs/1904.03949

[62] P. D. Alfano, V. P. Pastore, L. Rosasco, and F. Odone, "Fine-tuning or top-tuning? transfer learning with pretrained features and fast kernel methods," 2022. [Online]. Available: https://arxiv.org/abs/2209.07932

[63] I. S. E. M. Abramowitz, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing. page 1020.* New York: Dover, 1972.

[64] Z. Wei, "Implementation of deeppatch," https://github.com/Wsine/deeppatch, 2022.

[65] Z. Wei, H. Wang, Z. Yang, and W. Chan, "Sebox4dl: A modular software engineering toolbox for deep learning models," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 193–196.

[66] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Proceedings of Advances in Neural Information Processing Systems*, vol. 32. Curran Associates, Inc., 2019.

[67] L. Bottou, "On-line learning and stochastic approximations," in *Proceedings of In On-line Learning in Neural Networks*. Cambridge University Press, 1998, pp. 9–42.

[68] I. Loshchilov and F. Hutter, "SGDR: stochastic gradient descent with restarts," *CoRR*, vol. abs/1608.03983, 2016. [Online]. Available: http://arxiv.org/abs/1608.03983

[69] D. Hendrycks*, N. Mu*, E. D. Cubuk, B. Zoph, J. Gilmer, and B. Lakshminarayanan, "Augmix: A simple method to improve robustness and uncertainty under data shift," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=S1gmrxHFvB

[70] A. Jeddi, M. J. Shafiee, and A. Wong, "A simple fine-tuning is all you need: Towards robust deep learning via adversarial fine-tuning," 2020.

[71] E. Englesson and H. Azizpour, "Generalized jensen-shannon divergence loss for learning with noisy labels," in *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., vol. 34. Curran Associates, Inc., 2021, pp. 30 284–30 297. [Online]. Available: https://proceedings.neurips.cc/paper/2021/file/fe2d010308a6b3799a3d9c728ee74244-Paper.pdf

[72] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[73] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[74] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, September 2018.

[75] chenyaofo, "Pytorch cifar models," https://github.com/chenyaofo/pytorch-cifar-models, 2021.

[76] aaron xichen, "Implementation of pytorch-playground," https://github.com/aaron-xichen/pytorch-playground, 2020.

[77] tjmoon0104, "Implementation of pytorch-tiny-imagenet," https://github.com/tjmoon0104/pytorch-tiny-imagenet, 2020.

[78] F. AI, "Papers with code - cifar benchmark," 2022, accessed: 2022-1-6. [Online]. Available: https://paperswithcode.com/sota/image-classification-on-cifar-10?tag_filter=3

[79] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-100 (canadian institute for advanced research)," 2009. [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[80] A. Coates, A. Ng, and H. Lee, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, G. Gordon, D. Dunson, and M. Dudík, Eds., vol. 15. Fort Lauderdale, FL, USA: PMLR, 11–13 Apr 2011, pp. 215–223. [Online]. Available: http://cs.stanford.edu/~acoates/stl10

[81] M. A. mnmoustafa, "Tiny imagenet," 2017. [Online]. Available: https://kaggle.com/competitions/tiny-imagenet

[82] Wikipedia contributors, "Gaussian blur — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=Gaussian_blur&oldid=1027662266, 2021, [Online; accessed 2-September-2021].

[83] gaoxiang9430, "Sensei implementation," https://github.com/gaoxiang9430/sensei, 2020.

[84] Z. Zhong, Y. Tian, and B. Ray, "Understanding local robustness of deep neural networks under natural variations," in *Fundamental Approaches to Software Engineering*, E. Guerra and M. Stoelinga, Eds. Cham: Springer International Publishing, 2021, pp. 313–337.

[85] D. Rey and M. Neuhäuser, *Wilcoxon-Signed-Rank Test*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1658–1659. [Online]. Available: https://doi.org/10.1007/978-3-642-04898-2_616

[86] kentaroy47, "Implementation of vision-transformers-cifar10," https://github.com/kentaroy47/vision-transformers-cifar10, 2022.

[87] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=YicbFdNTTy

[88] S. A. Marhon, C. J. F. Cameron, and S. C. Kremer, *Recurrent Neural Networks*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–65. [Online]. Available: https://doi.org/10.1007/978-3-642-36657-4_2

[89] M. u. Hassan, N. R. Sabar, and A. Song, "Optimising deep learning by hyper-heuristic approach for classifying good quality images," in *Proceedings of Computational Science – ICCS 2018*, Y. Shi, H. Fu, Y. Tian, V. V. Krzhizhanovskaya, M. H. Lees, J. Dongarra, and P. M. A. Sloot, Eds. Cham: Springer International Publishing, 2018, pp. 528–539.

[90] M. Usman, D. Gopinath, Y. Sun, Y. Noller, and C. S. Pasareanu, "Nnrepair: Constraint-based repair of neural network classifiers," *CoRR*, vol. abs/2103.12535, 2021. [Online]. Available: https://arxiv.org/abs/2103.12535

[91] X. Xie, W. Guo, L. Ma, W. Le, J. Wang, L. Zhou, Y. Liu, and X. Xing, "Rnnrepair: Automatic rnn repair via model-based analysis," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 11 383–11 392.

[92] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *Proceedings of 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 397–409.

[93] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input prioritization for testing neural networks," in *Proceedings of 2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019, pp. 63–70.

[94] L. Meng, Y. Li, L. Chen, Z. Wang, D. Wu, Y. Zhou, and B. Xu, "Measuring discrimination to boost comparative testing for multiple deep learning models," in *Proceedings of 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 385–396.

[95] W. Shen, Y. Li, L. Chen, Y. Han, Y. Zhou, and B. Xu, "Multiple-boundary clustering and prioritization to promote neural network retraining," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 410–422.

[96] S. Bing, S. Jun, L. H. Pham, and S. Jie, "Causality-based neural network repair," in *2021 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*, 2022. [Online]. Available: https://arxiv.org/abs/2204.09274

[97] H. F. Eniser, S. Gerasimou, and A. Sen, "Deepfault: Fault localization for deep neural networks," in *Proceedings of Fundamental Approaches to Software Engineering*, R. Hähnle and W. van der Aalst, Eds. Cham: Springer International Publishing, 2019, pp. 171–191.

[98] X. Ren, B. Yu, H. Qi, F. Juefei-Xu, Z. Li, W. Xue, L. Ma, and J. Zhao, "Few-shot guided mix for dnn repairing," in *Proceedings of 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 717–721.

[99] K. O. Stanley, J. Clune1, and J. Lehman1, "Designing neural networks through neuroevolution," *Nature Communications*, vol. 11, 2020.

[100] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary Intelligence*, vol. 1, no. 1, pp. 47–62, Mar 2008.

[101] D. Hendrycks, "Cifar-10-c and cifar-10-p," Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2535967

[102] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[103] D. Zhao, Z. Xing, C. Chen, X. Xia, and G. Li, "Actionnet: Vision-based workflow action recognition from programming screencasts," in *Proceedings of 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 350–361.

[104] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proceedings of 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016, pp. 51–62.

[105] J. Chen, C. Chen, Z. Xing, X. Xia, L. Zhu, J. Grundy, and J. Wang, "Wireframe-based ui design search through image autoencoder," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 3, jun 2020.

[106] M. Havranek, C. Bernal-Cárdenas, N. Cooper, O. Chaparro, D. Poshyvanyk, and K. Moran, "V2s: A tool for translating video recordings of mobile app usages into replayable scenarios," in *Proceedings of 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021, pp. 65–68.

[107] Y. Li, S. Wang, and T. Nguyen, "Fault localization with code coverage representation learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 661–673.

[108] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 309–321.