# Plum: Exploration and Prioritization of Model Repair Strategies for Fixing Deep Learning Models†

Hao Zhang
Department of Computer Science
City University of Hong Kong
hzhang339@my.cityu.edu.hk

W.K. Chan‡
Department of Computer Science
City University of Hong Kong
wkchan@cityu.edu.hk

***Abstract*—The accuracy of DL models may not meet the user's expectations. To tackle this problem, existing work proposed diverse approaches, such as using more optimized training processes and training samples to evolve the model structure or parameters of such faulty DL models. In this paper, we present Plum, a novel hyperheuristic approach to fixing deep learning models. Plum generates a set of DL model candidates by applying low-level repair strategies. It then evaluates and prioritizes repair strategies based on their overall fixing effects exhibited by these model candidates and outputs a fixed DL model by applying the top-ranked repair strategy. We also formulate a novel repair strategy to show the compatibility of Plum in incorporating new repair strategies. The experiment on five DL models showed that Plum achieved improvements in test accuracy by 2.49% and 3.11% on the CIFAR-10 and CIFAR-100 datasets over the baselines and outperformed Apricot and MODE, two previous state-of-the-art deep learning repair techniques.**

***Keywords-Deep neural networks; Model evolution; Debugging; Model repair; Strategy prioritization; hyperheuristic***

## I. INTRODUCTION

A deep learning (DL) model *D* consists of a neural network architecture and a set of trained parameters [1]. For a classification task, It accepts a sample *x* (e.g., an image) as input, conducts an inference, and produces an output vector associated with a sequence of probabilities where each element in the output vector represents the likelihood of *x* belonging to that class. When there is an imperfection of a DL model that exposes a failure of the DL model, the DL model is called ***faulty***. For instance, if a model *D* should but does not correctly classify a sample *x*, then *D* is regarded as *faulty*.

Developers aim to test a DL model (denoted as *D*) on whether its predicted output same as the corresponding expected result (i.e., the ground truth of each given sample). They further want to repair the model if the model is faulty. For ease of our presentation, we refer to the output class with the highest predicted probability of the DL model *D* on inferencing a sample *x* as ***D*(*x*)**. If *D*(*x*) is the same as the expected result of *x*, then *x* is *correctly classified*, otherwise *misclassified*, and is called a ***correct test case***, otherwise a ***failing test case***.

† This research is supported in part by the GRF of Hong Kong RGC (project nos.11214116 and 11203517), the HKSAR ITF (project no. ITS/378/18), CityU MF_EXT (project no. 9678180), and a CityU SGS Conference Grant.

‡ The corresponding author.

Traditional software debugging techniques enable developers to localize and/or eliminate bugs contained in programs. Nevertheless, debugging a deep learning model requires addressing new challenges. Precisely localizing root causes of deep learning models misclassifying a sample is still not well understood [2]. As shown in prior studies [3][4][5], many DL models can hardly achieve 100% in test accuracy. As such, failing test cases at the inference time are not rare for many classification tasks. However, many model debugging studies [6][7][9][11] have shown that by modifying the trained parameters of a DL model with respect to some failing test cases, a resultant model with higher accuracy could be obtained.

For each of our presentations, we define a notion of fixing a DL model as follows. Suppose that we have two versions $D_{buggy}$ and $D_{repaired}$ of a DL model *D,* where testers repair $D_{buggy}$ to become $D_{repaired}$ with a criterion *C* (called a ***fixing task***) to quantify the imperfection of $D_{buggy}$ aiming to rectify. The version $D_{repaired}$ *resolves C* if *C* is no longer met by $D_{repaired}$. The difference between the two versions is a *fix* for $D_{buggy}$.

A model version, say $D_{buggy}$, may have more than one fixing task to resolve (e.g., two exemplified fixing tasks are to rectify failing test cases $x_1$ and $x_2$, respectively.) In practice, fixing $D_{buggy}$ may generate a model $D_{repaired}$ that resolves some but not all of the given fixing tasks. Thus, a viable goal for model fixing is to resolve as many given fixing tasks as possible. A typical side effect of resolving fixing tasks is to make $D_{repaired}$ incur additional fixing tasks (e.g., a sample $x_3$ is a failing test case of $D_{repaired}$ but a correct test case of $D_{buggy}$), where $D_{buggy}$ needs not to resolve. An effective fixing technique should resolve fixing tasks with a small side effect. An alternative practical goal for a fixing technique is to improve the test accuracy of a given model while able to resolve the given fixing tasks to some extent.

For brevity, we categorize existing deep learning fixing techniques (e.g., [6][7][8][9]) by two dimensions of how they use artifacts: (1) whether or not additional samples are required by a technique (e.g., [7] versus [8], data augmentation techniques [10] can also be categorized as introducing extra augmented samples), and (2) whether or not any additional neural network architecture is required in the fixing process (e.g., [6] versus [9]). Among existing fixing techniques (see review in Section V), Apricot [9] needs *neither* additional samples *nor* additional neural network architectures. The model repair approach explored by Apricot could be more generic than those techniques requiring additional samples or architecture as debugging assistances. We are interested in studying more generic kinds of

approaches and wonder how far such approaches can go. In this paper, we study along this direction.

Furthermore, to the best of our knowledge, all existing fixing work [6][7][9][11] proposed a set of effective and/or efficient repair strategies, and yet no proposed strategy excelled in all cases in their respective experiments. For instance, MODE [7] evaluated each layer to identify buggy neurons and further generated fixing patches for retraining. However, the fixing efforts could be limited on those datasets with complex features. In [9], Apricot proposed three fixing strategies, but none of which could outperform other strategies in all cases. DeepCorrect [6] fixed defects contained in the convolutional layers, ignoring other potential defects in other types of layers. In other words, the improper usage of fixing techniques brings little benefit in addressing the issues those techniques were initially developed for, causing unexpected costs.

We also observe that these existing techniques merely deem their generated models achieving the highest performance on their test datasets as their fixed models. They either ignore the performance of these models on the validation dataset (where the validation datasets are incorporated in their techniques or their repair strategies) or simply assume that a validation dataset is a test dataset, implicitly assuming that the observed performance on the validation dataset is consistent to the performance on a dataset unseen in their model training. As an analog, the situation is more like using a DL model with the highest recall to guide the model selection in each trial run, and among these selected model candidates across trial runs, the one with the highest performance on the test dataset is finally chosen.

In this paper, we propose a novel deep learning repair technique called Plum to address the above problems. It is built on top of Apricot. We firstly revisit Apricot as summarized below: Suppose that there is a faulty DL model $D_{buggy}$ produced by a training scheme $\mathcal{S}$ where each fixing task is a unique failing test case in the training dataset $T_o$ of $\mathcal{S}$. Apricot firstly utilizes $D_{buggy}$'s neural network architecture to generate a set of models, each of which is trained with $\mathcal{S}$ using a subset of $T_o$. For each fixing task, it divides these generated models into those resolving the task and the counterpart. It then computes the mean weights of these models in each of the two divided sets and adjusts $D_{buggy}$ by treating a scaled-down version of the abovementioned mean weights as the weight search direction to adjust $D_{buggy}$'s parameters. Apricot has three standalone repair strategies: one only takes either set of generated models to compute the corresponding mean weights, and the third compute both. Section II.C will introduce them in more detail.

Plum treats each repair strategy of Apricot as a low-level repair strategy. It also proposes a novel and effective low-level repair strategy. It applies each of them to fix the same faulty model $D_{buggy}$ to produce a fixed model candidate. It looks for a balance between the model performances on the validation and test datasets achieved by each low-level strategy by formulating a series of prioritization schemes to rank low-level strategies (not individual models). The top-ranked model of the top-ranked strategy is outputted as the fixed DL model.

We evaluate Plum on five representative DL models on the CIFAR-10 and CIFAR-100 datasets and compare it to Apricot and MODE. The result shows that Plum outperforms Apricot and MODE. It improves the model performance in terms of test accuracy by 1.70%-3.94% and 2.66%-4.10% on CIFAR-10 and CIFAR-100 compared to the baseline, indicating the effectiveness of Plum.

The main contribution of this paper is threefold: (1) This paper is the *first* work to present a novel hyperheuristic fixing technique that can effectively fix DL models and outperform the peer techniques at the same time. It also formulates a novel low-level repair strategy to show the possibility of Plum in embedding extra repair strategies. (2) It is also the *first* repair strategy prioritization technique to assess the consistency between fixed DL models achieved by model repair strategies. (3) It presents an evaluation on Plum to show its feasibility and effectiveness in fixing DL models.

The organization of the remaining sections is as follows. Section II revisits the preliminaries. Section III presents Plum and a novel repair strategy, followed by its evaluation in Section IV. Section V discusses closely related work. We conclude this work in Section VI.

## II. Preliminaries

### A. Deep Learning Models

Typically, a deep learning model [1] consists of two main parts: a model architecture (e.g., neurons, layers, connections between layers) and a set of parameters (e.g., weights) represented as a set of weight matrices. The model architecture can be generated automatically [12] or designed manually [13][14][15]. Iterative learning algorithms such as Stochastic Gradient Descent (SGD) or evolutionary algorithms [16] are often used to update model weights. The process of updating weights is usually realized by the forward and backward propagations. The *standard training scheme* refers to the typical process of training a DL model [17].

Typically, one or several datasets are used in training or evaluating a DL model. The set of input data instances (or samples) for training a DL model is called a *training dataset*, and the set of samples for evaluating the model is called a *test dataset*. A *validation dataset* is a set of samples for estimating the performance of the DL model and tuning hyperparameters. The best practice is to ensure the three datasets to be mutually exclusive. At the same time, in quite most experiments, the validation dataset is simply treated as the test dataset.

### B. Deep Learning Fixing and Debugging

Some studies [7][18][19] have shown that DL models are error-prone. In the literature, DL model bugs can be broadly classified into two types [7]: structural bugs and training bugs. Structural bugs refer to the structure of a DL model (including the type of layers, connections between layers, types, and allocations of activation function) being not well-designed. State-of-the-art DL models are usually complex and manually designed. Automatically fixing this type of bug could be challenging since the number of candidates is usually extremely huge, making the search space much larger than

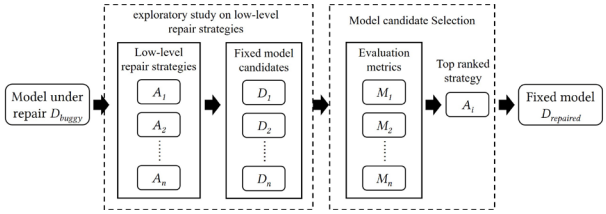


Figure 1. The overall process of Plum

fixing training bugs. As presented in Section I, DeepCorrect, although limited in scope, fixes this kind of bug.

Training bugs refer to some weights of the DL model being insufficiently trained due to the presence of blockers, e.g., biases or conflicting samples in the datasets used in the training process (which is known as the underfitting problem), or being exhaustively optimized due to improper settings or optimization techniques (which is known as the overfitting problem). Following [7], we refer to these weights as ill-trained weights in this paper. An insight is that ill-trained weights are most responsible for the misclassification of the deep learning model with respect to certain test inputs.

Locating these ill-trained weights precisely is challenging. State-of-the-art DL models usually contain millions of weights and parameters, and identifying which weights are responsible for the erroneous behaviors of the deep learning model precisely is still unreachable yet.

A relating approach to addressing this problem is to train a deep learning model over an additional dataset or to train the model with different weights initializations and then select the resultant model with the highest performance. Exposing more data inputs to the DL model is widely considered as a practical and popular approach to fixing ill-trained weights. Nonetheless, using more inputs for retraining requires more labeling efforts in data collection, and collecting sufficient inputs may be impractical in many cases.

### C. Revisit the Apricot Approach

Apricot [9] is an adaptation approach to fix deep learning models iteratively. The intuition of Apricot is that DL models trained on a small set of the training dataset can provide insights on adjusting weights of faulty DL models. Given a faulty DL model $D_{buggy}$ and a training dataset $T_0$, Apricot first generates a set of submodels $\Omega$, each of which is trained on a random subset of $T_0$. Those resultant DL models trained on the subsets of $T_0$ are referred to as ***reduced deep learning models* (*rDLMs*)**. For ease of understanding, we followed the same abbreviation in this paper. Then, it investigates each training sample $x$ and checks whether $D_{buggy}$ classifies $x$ correctly. If there is no misclassification, the algorithm continues to check the next input. Otherwise, it partitions $\Omega$ into two sets: a set of models $\Omega_c$ that each classifies $x$ correctly and a set of models $\Omega_f$ that each classifies $x$ incorrectly. Apricot then adjusts the weights of $D_{buggy}$ by utilizing $\Omega_c$ and $\Omega_f$ with three proposed strategies and further trains the weight-adjusted $D_{buggy}$ with $T_0$ for some epochs. After iterating all training samples, Apricot produces a model $D_{repaired}$ as the output.

**Algorithm 1** Low-Level Repair Strategy Exploration

Input: $D_{buggy} \leftarrow$ DL model to be debugged
$D_{init} \leftarrow$ initialized DL model to produce $D_{buggy}$
$T_o \leftarrow$ training dataset
$T_v \leftarrow$ validation dataset
$N_1 \leftarrow$ number of epochs for training rDLMs
$N_2 \leftarrow$ number of epochs for further training rDLMs
$m \leftarrow$ number of rDLMs to produce
$\Theta \leftarrow$ set of low-level repair strategies
$N_{r1} \leftarrow$ number of repair exploration attempts
$N_{r2} \leftarrow$ number of iterations in fixing
$N_{r3} \leftarrow$ number of epochs in fixing

Output: $\Phi \leftarrow$ set of fixed DL model candidates

**Procedure** Explore($D_{buggy}$, $D_{init}$, $T_o$, $N_1$, $N_2$, $m$)
**foreach** $i$ from 1 to $N_{r1}$ **do**
  $S = \{T_1, \ldots, T_m\} \leftarrow$ Reduce($T_o$, $m$)
  $\Omega = \{\}$
  **for** $T_i \in S$ **do**
    $D_i \leftarrow$ Clone($D_{init}$)
    $D_i \leftarrow$ Train($D_i$, $T_i$, $N_1$)
    $rDLM_i \leftarrow$ Train($D_i$, $T_o$, $N_2$)
    $\Omega = \Omega \cup \{ rDLM_i \}$
  $\Phi = \{\}$
  $S_c \leftarrow$ GetCorrectTestCase($T_o \cup T_v$, $D_{buggy}$)
  $S_f \leftarrow$ GetFailingTestCase($T_o \cup T_v$, $D_{buggy}$)
  **foreach** $\theta \in \Theta$ **do**
    $D_{\theta,i} = \theta$.Fix($S_c$, $S_f$, $\Omega$, $T_o$, $T_v$, $N_{r2}$, $N_{r3}$, $D_{buggy}$)
    $\Phi = \Phi \cup \{ D_{\theta,i} \}$
**return** $\Phi$

The three strategies formulated in Apricot are as follows. For each layer $l$ of $D_{buggy}$, the mean weight matrices (denoted as $A$ and $B$) of the corresponding layer of the models in $\Omega_c$ and $\Omega_f$ are computed. Suppose that the weight matrix of layer $l$ is $L$. The three strategies update the weights in $L$ by $L + \alpha A$, $L + \alpha A - \beta B$, and $L - \beta B$, respectively, where $\alpha$ and $\beta$ are learning rates. In this paper, we denote the three strategies are $\boldsymbol{A_1}$, $\boldsymbol{A_2}$, and $\boldsymbol{A_3}$, respectively.

## III. PLUM

### A. Overview

In this paper, we propose Plum. Plum is a novel hyperheuristic approach to fixing DL models. It is based on two observations. First, as we have presented in Section I, certain fixing strategies could only be effective in specified situations. Fixing different DL models on different learning tasks may require different strategies to achieve the best results. Thus, a single-strategy technique may be limited to a suboptimal solution in handling certain cases. Second, the model performances on the validation dataset and test dataset are independently assessed when determining a fixed model $D_{repaired}$. In machine learning, developers often conduct a process of trial and error, i.e., run the same training scheme a few times and pick the one with the best result as the final

**Algorithm 2** Low-Level Repair Strategy Prioritization

Input: $\Theta \leftarrow$ set of low-level repair strategies
$\Phi \leftarrow$ set of fixed DL model candidates
$T_v \leftarrow$ validation dataset
$T_t \leftarrow$ test dataset
$\mathfrak{T} \leftarrow$ the indicator vector

Output: $D_{repaired} \leftarrow$ fixed DL model

**Procedure** Prioritize($\Phi$,$T_v$, $T_t$, $\mathfrak{T}$)
1: **foreach** $\theta \in \Theta$ **do**
2: $\mathfrak{D}_\theta^v = \{ \text{Eval}(D_{x,i}, T_v) \mid D_{x,i} \in \Phi \text{ and } x = \theta \}$
3: $\mathfrak{D}_\theta^t = \{ \text{Eval}(D_{x,i}, T_t) \mid D_{x,i} \in \Phi \text{ and } x = \theta \}$
4: $a_\theta^v, m_\theta^v$ = average and max values in $\mathfrak{D}_\theta^v$
5: $a_\theta^t, m_\theta^t$ = average and max values in $\mathfrak{D}_\theta^t$
6: Rank $\Theta$ based on $a_\theta^v, m_\theta^v, a_\theta^t, m_\theta^t$ for all $\theta \in \Theta$ and the indicator vector $\mathfrak{T}$
7: $\theta_{top}$ = PickTop($\Theta$)
8: **return** $D_{x,i} \in \Phi$ such that $x = \theta_{top}$ and Eval$(D_{x,i}, T_t) = \max(v \in \mathfrak{D}_\theta^t)$

produced model. More specifically, in a trial run to produce a fixed model for a buggy model $D_{buggy}$, a fixed model candidate achieving the highest accuracy on the validation dataset in that trial run is often deemed as the output of that trial run. Among a set of such trial runs, the model candidate with the highest accuracy on the test dataset is selected as the fixed model $D_{repaired}$. Nonetheless, its accuracy on the validation dataset might not be one of the best few, posting a question mark on whether or not the fixed model has good generalization.

Plum consists of two phases: a phase for exploratory study on low-level repair strategy and a phase for fixed model candidate selection (as depicted in Figure 1). The first phase imports a set of low-level strategies. In our current design, there are three repair strategies of Apricot (see Section II.C), and to demonstrate that Plum can work with new strategies, we also design a novel low-level repair strategy (see Section III.C). Plum firstly generates a set of submodels as required by Apricot. Then, it explores each low-level repair strategy by applying it to produce a set of fixed model candidates for the same given faulty DL model $D_{buggy}$ for the given fixing task. (In this paper, every failing test case in the training or validation dataset of $D_{buggy}$ represents a fixing task.) Plum measures the accuracy achieved by each fixed model candidate on the validation and test datasets. In the second phase, Plum computes several performance evaluation metrics, including the average and maximum validation accuracies and the average and maximum test accuracies achieved by each low-level strategy. It then prioritizes the low-level strategies by considering the balance in performance across validation and testing (represented by the validation and test datasets). Such as, a top-ranked low-level strategy is identified. We are going to present more details about the prioritization schemes in the following subsection. Plum finally outputs the fixed model candidate achieving the highest test accuracy produced by the top-ranked strategy as the fixed model $D_{repaired}$ of $D_{buggy}$.

### *B. Low-Level Repair Strategy Exploration Phase of Plum*

The first phase of Plum consists of the generation of rDLMs and exploration performed by low-level repair strategies. The procedure to generate a set of rDLMs is adopted from the algorithm of Apricot, which is summarized as lines 3-8 in Algorithm 1. The remaining part of Algorithm 1 is the proposal of this paper.

Algorithm 1 firstly accepts as input a faulty DL model $D_{buggy}$ and its initialized version $D_{init}$ as well as the training dataset $T_o$, validation dataset $T_v$, the number $m$ of submodels to be produced, and the pair of numbers $N_1$ and $N_2$, representing the numbers of epochs to train a submodel using a subset of the training dataset and the whole training dataset, respectively. It also accepts a set of low-level repair strategies $\Theta$ and three more parameters $N_{r1}$, $N_{r2}$, and $N_{r3}$ as inputs. They are used to control the total number of trial attempts of each repair strategy, the number of fixing iterations, and subsequent further training on the model candidate being evolved for each fixing task in each trial attempt, respectively.

The DL model $D_{init}$ is a pretrained version of $D_{buggy}$. In standard training, a neural network architecture after initialization is trained on the training dataset $T_0$ with several epochs to produce a preliminarily trained version of $D_{buggy}$, which is denoted by $D_{init}$.

At lines 1-14, the algorithm iterates $N_{r1}$ times. In line 2, the algorithm generates a set of datasets $S = \{T_1, T_2, \ldots, T_m\}$, each of which being a subset of $T_o$, where $m$ is the number of DL models each being trained with such a dataset. To ensure that all samples in $T_o$ have been utilized, the constraint $T_o = \bigcup_{i=1}^{m} T_i$ should hold.

At lines 5-6, the algorithm trains a copy of the model $D_{init}$ for each dataset in $S$ for the number of epochs specified by the parameter $N_1$ so that the trained model recognizes the task encoded in the corresponding dataset. Then, at line 7, the model is further trained with the whole training dataset $T_0$ for a few more epochs (specified by $N_2$). The algorithm thus generates a set of trained models $\Omega$.

Then, at lines 10-11, the algorithm regroups the training and validation samples into the set of samples that the faulty model $D_{buggy}$ classifies each of them correctly (line 10) and into another set of samples that $D_{buggy}$ misclassifies each (line 11). These two sets are denoted as $S_c$ and $S_f$, respectively.

After that, the algorithm iterates all low-level repair strategies (lines 12-14). For each given low-level repair strategy $\theta$, the algorithm generates a fixed model candidate $D_{\theta,i}$ (line 13). These candidates are kept in the set $\Phi$ (line 14), which is initialized as an empty set at line 9.

In line 13, the function *Fix*() takes $S_c$ and $S_f$ as inputs. If the underlying strategy does not require a particular set of test cases, it can ignore the provided dataset. For instance, $A_1$ only requires $S_c$, and $A_3$ only requires $S_f$.

### *C. Strategy and Fixed Model Selection Phase of Plum*

The second phase of Plum is to assess the low-level repair strategies for consistency and quality and selects a promising fixed model candidate produced by that strategy as the output of Plum. Its algorithm is summarized in Algorithm 2.

**Algorithm 3** Model Repair Strategy

Input: $D_{buggy}$ ← faulty DL model
$S_c$ ← *set of correct test cases*
$S_f$ ← *set of failing test cases (as fixing tasks)*
$\Omega$ ← set of rDLMs
$T_o$ ← training dataset
$T_v$ ← validation dataset
$N_{r2}$ ← number of incremental fixing rounds
$N_{r3}$ ← number of epochs for brief retraining

Output: $D_{repaired}$ ← fixed model candidate

**Procedure** Fix($S_c$, $S_f$, $\Omega$, $T_o$, $T_v$, $N_{r2}$, $N_{r3}$, $D_{buggy}$)
1: $ACP$ ← CalACP($T_o$, $\Omega$)
2: $D_{repaired} = D_{buggy}$
3: **for** $i$ from 1 to $N_{r2}$ **do**
4: $D_{repaired}$ ← Patch($S_f$, $\Omega$, $ACP$, $T_v$, $D_{repaired}$)
5: $D_{repaired}$ ← Train($D_{repaired}$, $T_o$, $N_{r3}$)
6: $D_{repaired}$ ← Patch($S_c$, $\Omega$, $ACP$, $T_v$, $D_{repaired}$)
7: $D_{repaired}$ ← Train($D_{repaired}$, $T_o$, $N_{r3}$)
8: **return** $D_{repaired}$

**Procedure** Patch($S$, $\Omega$, $ACP$, $T_v$, $D_{repaired}$)
1: **foreach** $x$ **in** $S$ **do**
2: $y$ = *groundtruth of* $x$
3: $\Omega_c$, $\Omega_f$ ← Divide($\Omega$, $x$, $y$, $ACP$)
4: $w_c$, $w_f$ ← Select($\Omega_c$, $\Omega_f$)
5: $W$ ← Eq. (4) and Eq. (5)
6: $D_{repaired}$.set($W$)
7: Eval($D_{repaired}$, $T_v$)
8: **return** $D_{repaired}$

In Algorithm 2, line 1 iterates on each repair strategy explored by the first phase of Plum. For each such strategy θ, it evaluates the set of fixed model candidates produced by that strategy using the validation dataset (line 2) and the test dataset (line 3) in terms of accuracy achieved by each model $D_{\theta,i}$ on these two datasets. It then computes more metric values of the candidate model set, including the average and maximum accuracy achieved by the model set on each of the two datasets (lines 4 and 5).

Algorithm 2 then prioritizes the strategies in Θ (line 6). After strategy prioritization, the strategy with top priority $\theta_{top}$ is selected (line 7). Then, it returns the model candidate that achieves the highest test accuracy among candidates produced by strategy $\theta_{top}$ in the set Φ as the final fixed model $D_{repaired}$.

We are aware that a model that excels on the validation dataset may be relatively less accurate on the test dataset, and vice versa. We formulate the following prioritization scheme to explore the design space. Each strategy is assigned with a *score* as follows:

$$score(\theta) = \frac{[a_\theta^v, m_\theta^v, a_\theta^t, m_\theta^t] \cdot \mathfrak{T}^T}{\sum_{i=1}^{n} \mathfrak{T}_i} \quad (1)$$

where $a_\theta^v$, $m_\theta^v$, $a_\theta^t$, and $m_\theta^t$ are average and maximum values in $\mathfrak{D}_\theta^v$ and $\mathfrak{D}_\theta^t$, respectively. $\mathfrak{T}$ is an indicator vector indicating which values are used in calculating the score. For example, if $\mathfrak{T}$ is [1, 1, 1, 1], indicating that all values are involved, then $score(\theta) = (a_\theta^v + m_\theta^v + a_\theta^t + m_\theta^t)/4$. If $\mathfrak{T}$ is [1, 0, 1, 0], then $score(\theta) = (a_\theta^v + a_\theta^t)/2$.

All repair strategies are prioritized based on their scores in descending order. The repair strategy $\theta_{top}$ with the highest score is selected to repair the faulty DL model. Finally, the fixed DL model candidate $D_{x,i}$ where $x = \theta_{top}$ achieving the highest test accuracy produced by this top-ranked repair strategy is taken as the output. (A further generalization can allow $\theta_{top}$ to explore our model candidates to find better solutions.)

### *D. A Novel Low-level Repair Strategy for Plum*

To demonstrate that Plum can work beyond Apricot, we design a novel repair strategy summarized in Algorithm 3. Algorithm 3 consists of two procedures: Fix() and Patch(). In procedure Fix(), the algorithm firstly computes the *Average Classification Probability* (ACP) for each rDLM.

Our intuition is as follows. We observe that given a failing test case $x$, an rDLM may classify $x$ to a label same as $x$'s ground truth but without much confidence (i.e., with a low prediction probability). In this case, the rDLM may not retain clear features essential for the correct classification of samples similar to $x$ in general. Thus, using this rDLM to guide the adjustment direction of the weights of the faulty DL model may post an adverse effect on the fixing process.

To alleviate this issue, we propose to use the Average Classification Probability (ACP) to assess whether an rDLM with respect to a particular input is of good quality.

We recall that the SoftMax layer of a model is often used as the output layer of a DL model, and its outputs can be viewed as a sequence of probability values for the corresponding labels. Given a set of samples $S$ that an rDLM classify correctly, the ACP of the rDLM for the set $S$ is defined as follows:

$$P(x_k \in S) = \{p_1, p_2, \ldots, p_c\} \quad (2)$$

$$ACP_i = \frac{\sum_{k=1}^{N} P(x_k)_i}{N} \quad (3)$$

where $P(x_k)$ is a set of predicted probabilities of labels of the rDLM on $x_k$, and $c$ is the number of labels. Note that given an input $x$ that the rDLM classifies correctly, the DL model produces a set of probabilities and chooses the label with the highest probability as the predicted output. $P(x_k)_i$ refers to the probability of the $i$-th label of this rDLM on classifying $x_k \in S$, where a test case $x$ belongs to the set $S$. The parameter $N$ in Equation (3) represents the size of the dataset $S$ (i.e., $|S|$). In other words, this fixing strategy uses the performance of each rDLM on correctly classified samples used in the training dataset to determine a baseline quality level for each label of the rDLM.

At line 1, the ACPs of rDLMs are calculated via the function CalACP() based on Eq. (2) and Eq, (3). Then, the algorithm iterates $N_{r2}$ times (lines 3-7). In each iteration, procedure Patch() is called twice (lines 4 and 6), each of which taking different parameters and is followed by training $D_{repaired}$ with $N_{r3}$ epochs (lines 5 and 7). After the iteration, the procedure takes the fixed DL model $D_{repaired}$ as the output.

Procedure Patch() is as follows. It takes a set of samples $S$, the set of rDLMs Ω, the calculated $ACP$, the validation dataset $T_v$, and $D_{repaired}$ as inputs. At lines 1-7, the algorithm iterates $S$ by batch. In each iteration, the set Ω of rDLMs is divided into

the set $\Omega_c$ of rDLMs that each classifies $x$ correctly and the output probability of this rDLM on $x$ is higher than the ACP value of the class same as the ground truth of $x$, and the set $\Omega_f$ of remaining rDLMs (line 3).

Then, at line 4, two rDLMs are selected with respect to the currently failing test case $x$: one is taken from $\Omega_c$ achieving the highest predicted probability among rDLMs in $\Omega_c$ on the ground truth of $x$, and another is taken from $\Omega_f$ achieving the lowest predicted probability among rDLMs in $\Omega_f$ on the ground truth of $x$. The algorithm refers to these two rDLMs as $\omega_c$ and $\omega_f$, respectively.

At line 5, the weight $W$ of $D_{repaired}$ is generated via Eq. (4) and Eq. (5). Formally, the adjustment strategy is calculated as follows:

$$\Delta W^{(k)} = r_l I_c\left(W^{(k-1)} - W_c\right) - r_l I_f\left(W^{(k-1)} - W_f\right) \quad (4)$$

$$W^{(k)} = W^{(k-1)} - \Delta W^{(k)} \quad (5)$$

where $W^{(k)}$ is the adjusted weights after the $k$-th iteration. $W_c$ and $W_f$ are the weights of the two selected rDLMs $\omega_c$ and $\omega_f$, respectively. $r_l$ is a constant representing the learning rate. $I_c$ and $I_f$ are two indicators. If $\Omega_c$ and $\Omega_f$, are empty, respectively, then $I_c$ and $I_f$, respectively, are set to 0, indicating that the corresponding term will be ignored.

In our design, this strategy uses $S_f$ and $S_c$, in turn, to fix the faulty model. Our intuition is that the fixing strategy should firstly fix those test cases that $D_{repaired}$ misclassifies and then transfer robust features learned from rDLMs to $D_{repaired}$ by iterating those correct test cases. We also note that the strategy does not copy and override the original weights of the faulty model. Eq. (4) computes a small change (with the help of the learning rate $r_l$) and applies such small changes incrementally and iteratively in lines 3-7 of procedure Fix().

We refer to this proposed low-level repair strategy as $\boldsymbol{A_4}$.

## IV. Experiment

### A. Experimental Setup

We implemented Plum on top of Keras 2.3.1 [20] and TensorFlow 2.2.0 [21], which were two popular deep learning libraries written in Python. All experiments were performed on Windows 10, running on an Intel i7-8700K CPU, 32GB RAM, a 256GB SSD, and a single NVIDIA GeForce 2080-Ti GPU with the VRAM size of 11GB. Following [9], we ran each repair strategy and MODE five times.

**Datasets**. The experiments were conducted using two datasets: CIFAR-10 and CIFAR-100 [22]. The CIFAR-10 dataset contained 60,000 images in 10 classes (50,000 training samples and 10,000 test samples), with 5,000 training samples and 1,000 test samples per class. The CIFAR-100 dataset contained 60,000 images (50,000 training samples and 10,000 test samples) in 100 classes. Each class contains 600 images (500 training samples and 100 test samples).

For training the DL models and corresponding rDLMs, following [9], for each DL model trained for repair, we randomly selected 20% of the training dataset as the validation dataset. For the CIFAR-10 dataset and CIFAR-100 dataset, there were 40,000 training samples, 10,000 validation samples, and 10,000 test samples. We note that the validation dataset is usually constructed for fine-tuning parameters of the model or estimating test error, which is common in practice.

**Model Architecture**. Five DL models with representative network architectures were used in the experiment: ResNet-20 [13], ResNet-32 [13], MobileNet [14], MobileNetV2 [23] and DenseNet [15]. For ResNet-20 and ResNet-32, we used the existing implementations [24]. For MobileNet, MobileNetV2, and DenseNet, we used the models that are already embedded in Keras [25]. All these five DL models are commonly applied in practice and research studies [9][26][27]. We chose to use top-1 accuracy [28] to measure the performance of each DL model because we are in the debugging of DL models for correctness.

**Hyperparameters**. We followed the popular settings (e.g., [29][30]) for training DL models on the CIFAR-10 and CIFAR-100 datasets. The hyperparameters were set as follows. The size of batches for training DL models was set to 128. The stochastic gradient descent (SGD) optimizer was applied with a learning rate of 0.01 and a momentum of 0.9 [31]. We note that SGD has been widely applied in existing experiments and in practice [11][29][30][32]. The categorical cross-entropy loss was applied as the loss function, which was popular for training DL models on classification learning tasks.

In Algorithm 1, the pre-training epochs for initializing the DL model, i.e., $N_1$, were set to 10. $N_2$ and $N_3$ were set to 50 and 190, respectively. That is, the DL models are trained with 10 epochs for coarse initialization and trained with 200 epochs in total to generate $D_{buggy}$, which is a typical setting in practice [29][30]. For training rDLMs, we note that rDLMs are trained on small subsets of the original training dataset, and it requires much fewer epochs to make rDLMs converge. $N_{r1}$, $N_{r2}$, and $N_{r3}$ were set to 5, 2, and 3, respectively. Our rationale is that we wish to limit the computational overheads by limiting the number of iterations to a small number of iterations (compared to the original training process) for exploration study and repair strategies. Compared to the number of epochs for training DL models, the fixing efforts are relatively minimal.

In Algorithm 2, the indicator vector $\mathfrak{T}$ is [1, 1, 1, 1], meaning that all the computed metrics are used by default. We will study the effects of other indicator vectors in Section IV.B.

**Subsets of the training dataset**. We followed the subset division procedure made in the previous experiment [9] to produce the training datasets to train rDLMs where the size of each such subset was 20% of the training dataset (i.e., 10,000 training samples). Each sample in the training dataset was indexed. The separation of the training dataset into subsets in Algorithm 1 was as follows. Suppose that a training dataset is a sequence of samples $TD$. The $i$-th subset of the training dataset is the samples in $TD$[i * 2000] to $TD$[$2000i + 0.2|T| - 1$], where $|T|$ is the size of the original training dataset.

The descriptive statistics of the faulty DL models are shown in TABLE I and TABLE II. Columns 2 and 5 present the number of epochs for pre-training each DL model and the total number of epochs for training each DL model. Column 3 shows the number of parameters in each DL model. Column 4 represents the time for training the DL model. Columns 6-8 show the accuracies over the training dataset, the validation dataset, and the test dataset, respectively. Column 9 shows the differences between training accuracy and test accuracy, in the

TABLE III DESCRIPTIVE SUMMARY OF DL MODELS ON THE CIFAR-10 DATASET

| Model ($D_{buggy}$) | Pre-training epochs | Num. of parameters | Training time | Training epochs | Training acc. | Validation acc. | Test acc. |
|---|---|---|---|---|---|---|---|
| ResNet-20 | 10 | 0.27M | 40m48s | 200 | 95.02% | 85.70% | 85.80% |
| ResNet-32 | 10 | 0.47M | 54m26s | 200 | 96.29% | 86.13% | 86.20% |
| MobileNet | 10 | 3.24M | 41m34s | 200 | 97.95% | 75.35% | 75.83% |
| MobileNetV2 | 10 | 2.27M | 1h1m5s | 200 | 94.99% | 76.68% | 76.98% |
| DenseNet | 10 | 7.05M | 2h14m15s | 200 | 99.56% | 80.43% | 80.81% |

TABLE IV DESCRIPTIVE SUMMARY OF DL MODELS ON THE CIFAR-100 DATASET

| Model ($D_{buggy}$) | Pre-training epochs | Num. of parameters | Training time | Training epochs | Training acc. | Validation acc. | Test acc. |
|---|---|---|---|---|---|---|---|
| ResNet-20 | 10 | 0.28M | 41m27s | 200 | 75.99% | 55.09% | 55.43% |
| ResNet-32 | 10 | 0.48M | 56m33s | 200 | 82.23% | 55.20% | 55.42% |
| MobileNet | 10 | 3.33M | 41m45s | 200 | 97.75% | 36.26% | 37.00% |
| MobileNetV2 | 10 | 2.39M | 1h3m49s | 200 | 96.95% | 37.00% | 37.53% |
| DenseNet | 10 | 7.14M | 2h21m39s | 200 | 98.70% | 45.77% | 45.97% |

TABLE I DESCRIPTIVE SUMMARY OF RDLMS ON THE CIFAR-10 DATASET

| Model | Total training epochs | Training Time for each rDLM | Avg. Training acc. | Avg. Validation acc. | Avg. Test acc. |
|---|---|---|---|---|---|
| ResNet-20 | 50 | 2m44s | 76.14% | 72.76% | 72.91% |
| ResNet-32 | 50 | 4m07s | 77.51% | 73.75% | 73.76% |
| MobileNet | 50 | 2m48s | 64.84% | 58.66% | 59.30% |
| MobileNetV2 | 50 | 4m37s | 57.11% | 54.19% | 54.57% |
| DenseNet | 50 | 10m47s | 76.97% | 68.78% | 69.55% |

TABLE II DESCRIPTIVE SUMMARY OF RDLMS ON THE CIFAR-100 DATASET

| Model | Total training epochs | Training Time for each rDLM | Avg. Training acc. | Avg. Validation acc. | Avg. Test acc. |
|---|---|---|---|---|---|
| ResNet-20 | 50 | 2m43s | 36.12% | 32.08% | 32.24% |
| ResNet-32 | 50 | 4m02s | 37.67% | 32.61% | 32.98% |
| MobileNet | 50 | 2m50s | 37.14% | 23.26% | 23.59% |
| MobileNetV2 | 50 | 4m45s | 12.29% | 11.23% | 11.03% |
| DenseNet | 50 | 10m16s | 42.42% | 28.73% | 29.21% |

range of 9.22%-22.12% on the CIFAR-10 dataset and 20.56%- 60.75% on the CIFAR-100 dataset.

TABLE III and TABLE IV show the descriptive statistics of rDLMs. Column 2 presents the total number of epochs (including pre-training epochs) for training rDLMs. Column 3 shows the average time for training one rDLM, which is much less than training the corresponding DL model. Columns 4-6 show the mean training, validation, and test accuracies over five runs, respectively, which are much lower than the faulty DL models, as expected. Column 7 shows the differences in test accuracy between rDLMs and the corresponding DL models. It is evident that the accuracies of rDLMs are much lower than that of the corresponding DL model ranging from 11.91% to 22.98%, with an average of 15.57% on the CIFAR-10 dataset, and 14.28%-27.59% with an average of 21.07% on the CIFAR-100 dataset. The rDLMs of MobileNetV2 have the most apparent drops in test accuracy in both CIFAR-10 and CIFAR-100 datasets by 22.98% and 27.59%, respectively. We recall that these rDLMs were trained on subsets of the original training dataset. It is quite as expected that they did not achieve as high accuracy as the DL model under repair. The training accuracies, validation accuracies, and test accuracies of rDLMs are more or less similar to one another, indicating that rDLMs do not suffer from serious overfitting problems.

**Fixing Task**. We followed previous experiments [7][9] to define a fixing task as a set of failing test cases of a faulty DL model. All failing test cases in the training and validation datasets were included as the fixing task in the experiment. Readers can read from TABLE I that in our experiment, each DL model produced by the standard training was associated with at least one failing test case, and thus is a faulty model.

**Comparing Plum with Apricot [9] and MODE [7]**. We implemented Apricot and MODE. In [9], Apricot formulates three strategies to adjust the parameters of DL models. We refer to these three strategies as $A_1$, $A_2$, and $A_3$, and they have been reviewed in Section II.C.

Given a faulty DL model $D_{buggy}$, MODE [7] conducts a round of forward-layer analysis: For each layer, a feature model is constructed by extracting layers of $D_{buggy}$ from the input layer to the selected layer, followed by a fully connected layer. Weights of this fully connected layer are deemed as the degree of importance of the corresponding features. Then this feature model is trained and evaluated. If the similarity between outputs of this feature model and $D_{buggy}$ is lower than a predetermined threshold, then this selected layer is marked to be fixed. Given a failing test case $x$, a heat map is constructed for recognizing the degree of importance of features. For each classification category, MODE recognizes features responsible for correct/incorrect classification via differential analysis on heat maps between correctly and incorrectly classified test cases. Then test cases to which these features are most sensitive to them are selected for training $D_{buggy}$ to generate the fixed DL model $D_{repaired}$.

Our implementation of MODE was based on the source code in [55] with a minor change: In MODE, an original dataset was split into four parts: the training dataset, the validation dataset, the bug fixing dataset, and the test dataset. The training dataset and the validation dataset are used to train the model, and the bug fixing dataset is used to generate inputs for training and fixing the model. The test dataset is used for evaluating the model. In our experiment, we used the DL models produced by the standard training for Plum, Apricot, and MODE to fix. We also used the validation dataset (see above) as the bug fixing dataset in MODE.

To make the comparison fair, the case of the highest improvements in test accuracy achieved by Apricot and MODE are used in comparison with Plum. Following [9], we repeated Apricot five times on each faulty DL model to obtain the results. Recall that a new low-level repair strategy is

TABLE V THE FIXING RESULTS OF APRICOT, MODE, AND PLUM ON THE CIFAR-10 DATASET

| Model ($D_{repaired}$) | Improvements in test accuracy | | | | |
|---|---|---|---|---|---|
| | Apricot | | | MODE | Plum |
| | $A_1$ | $A_2$ | $A_3$ | | |
| ResNet-20 | 1.98% | **2.04%** | 0.37% | –2.82% | **2.04%** |
| ResNet-32 | **3.94%** | 3.80% | 1.25% | –3.38% | **3.94%** |
| MobileNet | **1.97%** | 1.90% | 1.14% | –4.50% | **2.52%** |
| MobileNetV2 | 1.96% | **2.35%** | 1.06% | –3.83% | 2.23% |
| DenseNet | 1.53% | **1.70%** | 0.70% | –5.01% | **1.70%** |
| Average | 2.28% | 2.36% | 0.90% | –3.91% | **2.49%** |

TABLE VI THE FIXING RESULTS OF APRICOT, MODE, AND PLUM ON THE CIFAR-100 DATASET

| Model ($D_{repaired}$) | Improvements in test accuracy | | | | |
|---|---|---|---|---|---|
| | Apricot | | | MODE | Plum |
| | $A_1$ | $A_2$ | $A_3$ | | |
| ResNet-20 | 2.38% | 3.00% | 0.43% | –4.82% | **3.00%** |
| ResNet-32 | 3.89% | 3.16% | 2.47% | –6.61% | **4.10%** |
| MobileNet | 1.97% | 2.18% | 0.96% | –16.90% | **2.66%** |
| MobileNetV2 | 2.62% | **2.89%** | 1.74% | –18.71% | **2.89%** |
| DenseNet | **2.88%** | 2.39% | 0.60% | –21.70% | **2.88%** |
| Average | 2.75% | 2.72% | 1.24% | -13.75% | **3.11%** |

presented in Section III.D. We also ran this strategy five times. Then, the results of these four strategies are aggregated to form the result of Plum. Similarly, we repeated the experiment on MODE five times. Note that for Apricot and MODE, only models with the highest improvements in test accuracy were used for comparison.

## B. Results and Data Analysis

### 1) Comparing Plum to Apricot and MODE

The results of DL models after applying Apricot, MODE, and Plum on the CIFAR-10 and CIFAR-100 datasets are shown in TABLE V and TABLE VI. In these two tables, columns 2-4 represent the increases in test accuracy achieved by $A_1$, $A_2$, and $A_3$ of Apricot, respectively. Columns 5-6 show the increases in test accuracy achieved by MODE and Plum. Numbers in bold represent the highest increases in test accuracy across all methods.

From the two tables, $A_1$, $A_2$, and $A_3$ of Apricot achieved increases in test accuracy by 1.53% to 3.94%, 1.70% to 3.80%, 0.70% to 1.25% on the CIFAR-10 dataset, and by 1.97% to 3.89%, 2.18% to 3.16%, 2.65% to 4.10% on the CIFAR-10 dataset, respectively. MODE failed to improve the test accuracy of all ten model-dataset combinations (and we have analyzed the underlying limitation of MODE and are going to discuss them in subsection *C*). Plum achieved improvements in test accuracy by 1.70% to 3.94% and 2.66% to 4.10% on the two datasets, respectively. The results showed that Plum outperformed or was as effective as Apricot and MODE in improving the test accuracy of DL models in all model-dataset combinations, and in most cases, was more effective.

### 2) Effects of prioritization schemes

Recall that in Algorithm 2, repair strategies are prioritized based on their average values of $a_\theta^v$, $m_\theta^v$, $a_\theta^t$, and $m_\theta^t$ (calculated by Eq. (1)) in descending order. To evaluate the effectiveness of the proposed prioritization scheme, for comparison purposes, we enumerate the possible variants of $\mathfrak{T}$ that include both validation and test datasets side and only the validation dataset side. We do not include the variants that only consider the test dataset because the sole purpose of different fixing techniques (e.g., MODE and Apricot) are to be guided by a validation dataset rather than a test dataset.

- **Avg-Avg**: We set $\mathfrak{T}$ to [1, 0, 1, 0]. This prioritization scheme computes the average of the average validation accuracy and average test accuracy achieved by a repair strategy. It prioritizes a repair strategy ahead of the other repair strategy if the computed average of the former strategy is larger than that of the latter. In the case of ties, it ranks the one with higher average test accuracy first.
- **Avg-Max:** We set $\mathfrak{T}$ to [1, 0, 0, 1]. This prioritization scheme is the same as Avg-Avg, except that the average test accuracy achieved by a repair strategy is replaced by the maximum test accuracy achieved by the same repair strategy.
- **Max-Avg:** We set $\mathfrak{T}$ to [0, 1, 1, 0]. This prioritization scheme is the same as Avg-Avg, except that the average validation accuracy achieved by a repair strategy is replaced by the maximum validation accuracy achieved by the same repair strategy.
- **Max-Max:** We set $\mathfrak{T}$ to [0, 1, 0, 1]. This prioritization scheme is the same as Max-Avg, except that the average test accuracy achieved by a repair strategy is replaced by the maximum test accuracy achieved by the same repair strategy.
- **Avg**: We set $\mathfrak{T}$ to [0, 0, 1, 0]. This prioritization scheme ranks repair strategies in descending order of average validation accuracy.
- **Max**: We set $\mathfrak{T}$ to [0, 0, 0, 1]. This prioritization scheme ranks repair strategies in descending order of maximum validation accuracy.

Recall that in Plum, the repair strategy is prioritized based on their $a_\theta^v$, $m_\theta^v$, $a_\theta^t$, and $m_\theta^t$, i.e., the average and maximum values of validation accuracy and test accuracy. For Avg-Avg, Avg-Max, Max-Avg, and Max-Max, we used the pair of $a_\theta^v$ and $a_\theta^t$, $a_\theta^v$ and $m_\theta^t$, $m_\theta^v$ and $a_\theta^t$, $m_\theta^v$ and $m_\theta^t$, to prioritize repair strategies. For Avg and Max, we used $a_\theta^v$ and $m_\theta^v$ only to prioritize repair strategies. Thus, the first four variants of $\mathfrak{T}$ still maintain a balance between seen samples (when guiding the model repair process) and unseen samples, albeit lesser than the default repair strategy of Plum.

The test accuracy improvements made by each variant over the faulty DL models are summarized in TABLE VII and TABLE VIII. The best result on each model-dataset combination is highlighted. The above six prioritization schemes achieved average improvements in test accuracy by 2.30%, 2.30%, 2.38%, 2.38%, 2.30%, 2.38% on the CIFAR-10 datasets, and 3.03%, 3.03%, 2.93%, 2.93%, 3.03%, 2.93% on the CIFAR-100 dataset, respectively. Plum achieved average improvements in test accuracy by 2.49% and 3.11% on the two datasets and outperformed these six variants.

### 3) Effects of individual low-level repair strategy in Plum

Recall that in the previous experiment, Plum has the three repair strategies of Apricot ($A_1$, $A_2$, and $A_3$) and the repair

TABLE VII RESULTS OF DIFFERENT PRIORITIZATION SCHEMES ON THE CIFAR-10 DATASET

| Model | Improvements in test accuracy | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Avg-Avg** | **Avg-Max** | **Max-Avg** | **Max-Max** | **Avg** | **Max** | **Plum** |
| ResNet-20 | 1.87% | 1.87% | 1.87% | 1.87% | 1.87% | 1.87% | **2.04%** |
| ResNet-32 | 3.59% | 3.59% | 3.59% | 3.59% | 3.59% | 3.59% | **3.94%** |
| MobileNet | 1.97% | 1.97% | 1.97% | 1.97% | 1.97% | 1.97% | **2.52%** |
| MobileNetV2 | 2.35% | 2.35% | **2.79%** | **2.79%** | 2.35% | **2.79%** | 2.23% |
| DenseNet | 1.70% | 1.70% | 1.70% | 1.70% | 1.70% | 1.70% | **1.70%** |
| Average | 2.30% | 2.30% | 2.38% | 2.38% | 2.30% | 2.38% | **2.49%** |

TABLE VIII RESULTS OF DIFFERENT PRIORITIZATION SCHEMES ON THE CIFAR-100 DATASET

| Model | Improvements in test accuracy | | | | | | |
|---|---|---|---|---|---|---|---|
| | **Avg-Avg** | **Avg-Max** | **Max-Avg** | **Max-Max** | **Avg** | **Max** | **Plum** |
| ResNet-20 | **3.00%** | **3.00%** | 2.54% | 2.54% | **3.00%** | 2.54% | **3.00%** |
| ResNet-32 | **4.10%** | **4.10%** | **4.10%** | **4.10%** | **4.10%** | **4.10%** | **4.10%** |
| MobileNet | **2.66%** | **2.66%** | **2.66%** | **2.66%** | **2.66%** | **2.66%** | **2.66%** |
| MobileNetV2 | 2.72% | 2.72% | 2.72% | 2.72% | 2.72% | 2.72% | **2.89%** |
| DenseNet | 2.65% | 2.65% | 2.65% | 2.65% | 2.65% | 2.65% | **2.88%** |
| Average | 3.03% | 3.03% | 2.93% | 2.93% | 3.03% | 2.93% | **3.11%** |

strategy presented in Section III.D (denoted as $A_4$) as its low-level repair strategies. To evaluate the contribution of each low-level repair strategy to Plum, we counted the number of times that each strategy is selected by Plum. The results are shown in TABLE IX and TABLE X. Columns 2-5 represent the numbers of times that the corresponding DL models were fixed by strategies $A_1$, $A_2$, $A_3$, and $A_4$, respectively. The last two rows reported the number of times that the selected strategy is the best one and the number of times that the corresponding strategy is selected.

In each table, $A_2$ and $A_4$ are selected twice, and $A_1$ is selected once in fixing all five DL models. The results indicate that the repair strategies $A_1$, $A_2$, and $A_4$ are more preferable in fixing DL models by Plum. Besides, they also show that our new repair strategy $A_4$ is competitive to other strategies of Apricot, and no strategy among $A_1$ to $A_4$ excels in all cases.

Furthermore, as shown in the last two rows in TABLE IX and TABLE X, Plum selected the most promising fixing strategies in most cases (i.e., 9 out of 10 model-dataset combinations), indicating the effectiveness of Plum in identifying effective fixing strategies.

### C. Further Analysis

The experimental results have shown that Plum achieved increases in test accuracy by 2.49% and 3.11% on the CIFAR-10 and CIFAR-100 datasets, respectively. Furthermore, Plum outperformed Apricot and MODE, indicating its effectiveness in improving the test accuracy of the model.

Recall that the highest average improvements in test accuracy on ResNet-20 and ResNet-32 are 2.04% and 3.94% on the CIFAR-10 dataset, and 3.00% and 4.10% on the CIFAR-100 dataset. As to be discussed below, by comparing with the results in the literature, we consider that our method is effective in fixing ResNet models.

Take the CIFAR-10 dataset, for example. In [13], models ResNet-20, -32, -44, -56, and -110 achieved the test accuracy of 91.25%, 92.49%, 92.83%, 93.03% and 93.57%, respectively. The differences in test accuracy between consecutive pairs of ResNet were 1.24%, 0.34%, 0.20%, and 0.54%, respectively. It showed that the increase in test accuracy of ResNet strongly relied on adding more residual layers, and yet the margin of the increase was still minimal. For instance, by increasing the number of residual layers from 20 to 110 (represented by ResNet-20 and -110, respectively), the difference in test accuracy was less than 2.0%. Similar evidence can be found on the official website of CIFAR10-ResNet [33], on which ResNet-20 and -32 achieved the test accuracy of 92.16% and 92.46%, respectively. The difference was 0.30% between the two models on this dataset, and that between ResNet-30 and ResNet-110 is 0.19%. From TABLE V and TABLE VI**,** the improvements from ResNet-20 to ResNet-32 produced by Plum are, in fact, large.

In our experiment, Plum is more effective than Apricot. In our implementation, Apricot is formulated as three low-level fixing strategies (i.e., $A_1$, $A_2$, and $A_3$), and, on the contrary, Plum is formulated as a high-level hyperheuristic strategy to identify the most promising low-level fixing strategy. As shown in TABLE V and TABLE VI, Plum selects strategies that achieved the highest increase in test accuracy successfully in 9 out of 10 model-dataset combinations. In comparison, strategies $A_1$ and $A_2$ of Apricot achieved the highest increase in test accuracy 3 and 4 out of 10 model-dataset combinations, indicating the effectiveness of Plum in identifying promising low-level fixing strategies.

However, the margin between Plum and Apricot is not large in the experiment. The reason is that three out of four strategies in Plum are the strategies of Apricot. We tend to believe that by using more strategies formulated by other works and including more model-dataset combinations, the difference in margin between the two techniques will be more apparent. Nonetheless, the experiment clearly shows that no strategy can excel in all cases, showing that Plum's approach to exploring the repair strategy space and prioritizing repair strategies is a method to select a promising low-level fixing strategy dynamically.

We note that Plum is not as efficient as Apricot. It is because Plum explores all low-level fixing strategies and then selects the most promising one, but other techniques like

TABLE IX NUMBER OF TIMES OF STRATEGIES PICKED BY PLUM ON THE CIFAR-10 DATASET

| Model | Low-level repair strategy | | | |
|---|---|---|---|---|
| | $A_1$ | $A_2$ | $A_3$ | $A_4$ |
| ResNet-20 | 0 | 1 | 0 | 0 |
| ResNet-32 | 1 | 0 | 0 | 0 |
| MobileNet | 0 | 0 | 0 | 1 |
| MobileNetV2 | 0 | 0 | 0 | 1 |
| DenseNet | 0 | 1 | 0 | 0 |
| Best Selection | 1 | 2 | 0 | 1 |
| Total | 1 | 2 | 0 | 2 |

TABLE X RESULTS OF DIFFERENT PRIORITIZATION SCHEMES ON THE CIFAR-100 DATASET

| Model | Low-level repair strategy | | | |
|---|---|---|---|---|
| | $A_1$ | $A_2$ | $A_3$ | $A_4$ |
| ResNet-20 | 0 | 1 | 0 | 0 |
| ResNet-32 | 0 | 0 | 0 | 1 |
| MobileNet | 0 | 0 | 0 | 1 |
| MobileNetV2 | 0 | 1 | 0 | 0 |
| DenseNet | 1 | 0 | 0 | 0 |
| Best Selection | 1 | 2 | 0 | 2 |
| Total | 1 | 2 | 0 | 2 |

Apricot only need to predetermine one fixing strategy. However, it is challenging to know which fixing strategy is the most promising one before running it. It is applicable to apply Plum other than applying other low-level fixing strategies independently.

In the experiment, we note that the improvements in validation accuracy and test accuracy are consistent. However, as shown in columns 6-7 of TABLE VII and TABLE VIII, if we only consider the improvement in validation accuracy or test accuracy, it is likely that the most promising fixing strategy will not be considered. It is necessary to consider validation and test accuracies collectively.

In our experiment, MODE produced discouraging results. We investigated the underlying reason. We observed that MODE selects certain samples for fixing detected buggy neurons. However, MODE simply takes the average of the samples with the same category to identify the essential features for that class. This operation might work if the dataset is relatively regular and straightforward, such as the MNIST dataset [34]. However, it might not be applicable if those essential features for the correct classification are placed in different positions of the image, such as the ImageNet dataset [47]. As shown in the experiment, MODE suffered from a more severe loss on the CIFAR-100 dataset compared to its performance on the CIFAR-10 dataset. Furthermore, MODE might have destroyed connections among non-adjacent layers when extracted feature models from the faulty DL models in its algorithm, which made the original weights to activate neurons of these connection-destroyed layers problematic with respect to the original faulty DL models.

We did not compare Plum with other debugging techniques like DeepCorrect. We note that Plum has provided a hyperheuristic debugging framework, and other debugging techniques could be formulated as one of the low-level fixing strategies in Plum. It would be interesting to include other techniques in Plum, and we leave it as future work.

Owing to the page limit, we did not report how well Plum handles the fixing tasks, such as how many failing test cases were turned into correct ones and vice versa after fixing. We leave the report of the detailed analysis in our future work.

### D. Threats to Validity

We implemented all fixing techniques on top of Keras and TensorFlow, which were two commonly applied deep learning libraries in practice. It was well-known that these tools contain bugs. Moreover, we used existing implementations of DL models to avoid our biases in implementation. Despite that these model implementations were widely used in various experiments, they may contain bugs. We did not observe any abnormalities in the experiment.

We evaluated different techniques on five CNN-based models for two image classification tasks. Our findings may not generalize to other types of deep learning models and learning tasks. We only used one kind of fixing task in the experiment. Including more fixing tasks can generalize the result further.

In this paper, we did not further analyze the effects of fixing efforts made by Plum, i.e., the number of completed fixing tasks and the side effects of fixing tasks. Another issue is that Plum explores all low-level fixing strategies and selects the most promising one for fixing, which would make Plum less efficient than individual fixing strategies. We would like to study these issues in future work.

Plum has many parameters to constrain the number of models to be explored. All of the input parameters used by the low-level strategies via θ.Fix() can be directly extracted from the standard training scheme object of the low-level strategies.

## V. RELATED WORK

### A. Debugging Deep Neural Network Models

We have reviewed and evaluated both Apricot [9] and MODE [7] in previous sections. In this section, we review the other closely related debugging work.

Eniser et al. [11] proposed DeepFault to fix deep learning models. For each neuron in a DL model, DeepFault investigates if the neuron is activated with respect to the correct or incorrect behavior of the model. It assigns each neuron with a suspicious value. For those neurons with higher suspicious values, DeepFault synthesizes inputs guided by the gradients of those neurons and corrects the behaviors of the model by retraining on the synthesized inputs.

Borkar et al. [6] proposed DeepCorrect to fix filters in deep learning models to enhance the robustness against image distortions. Given an input $x$ and a distorted $x'$, DeepCorrect investigates the outputs of filters in each convolutional layer. For each filter, suppose the outputs of the filter after feeding $x$ and $x'$ are $O$ and $O'$, respectively. DeepCorrect substitutes $O'$ with $O$ and evaluates if the predicted probability improves. A correction priority for each filter is calculated. For those filters with higher susceptibility, a correction unit that is built based on a residual function is added for making filters generate $O$ rather than $O'$. Their experiments show that DeepCorrect can improve the robustness of the deep learning model effectively.

Wang et al. [8] proposed RobOT to improve the robustness of DL models. RobOT proposed two evaluation metrics called Zero-Order Loss (ZOL) and First-Order Loss (FOL). It utilizes these two metrics to evaluate if the given sample has the potential to improve the robustness of the model. A fuzzing technique guided by FOL is also proposed to generate fixing patches for retraining DL models.

### B. Repair Strategy Prioritization

A key aspect of Plum is to prioritize repair strategies for fixing deep learning models. To the best of our knowledge, Plum is the first work in this area. However, prioritization of strategies has between widely studied in software testing and debugging of traditional software.

Byun et al. [35] proposed to utilize information derived from the process of feedforward and backward process to evaluate if the given test input has a higher probability to reveal weaknesses or misbehaviors of the DL model. Their work compared a set of evaluation metrics for prioritizing test inputs such as SoftMax outputs, Bayesian Uncertainty, and Input Surprise. The experiments showed that test case prioritization could help reduce efforts for testing DL models.

Feng et al. [36] proposed DeepGini, an approach to prioritize test inputs for DL models. Given a test input, DeepGini collects the inference results of the DL model and measures the likelihood of the input being misclassified. Those test inputs that are more likely to be misclassified have a higher priority to be tested.

Moreover, in DeepFault and DeepCorrect, neurons and convolution filters are prioritized according to their fault suspiciousness. Plum prioritizes debugging strategies rather than artifacts in DL models, which makes Plum novel.

### C. Hyperherustics

Hyperheuristic is a method of automatically selecting one heuristic that is most suitable for solving the target problem from multiple ones. For each heuristic, there could be certain strengths and weaknesses. The key idea of hyperheuristics is to combine strengths and compensate for weaknesses in those heuristics to generate better solutions.

Zhang et al. [37] proposed AutoTrainer, a DNN training monitoring and automatic repairing technique. AutoTrainer embeds a set of state-of-the-art repair solutions. During the training process of a DL model, AutoTrainer monitors the status of the model and detects potential problems such as vanishing gradient, exploding gradient, dying ReLU, and oscillating loss. For each identified problem, AutoTrainer selects and applies the most suitable solution for fixing.

Hassan et al. [38] proposed a hyperheuristic approach to optimize parameters of DL models, which consists of a high-level strategy and low-level heuristics. The high-level strategy evaluates the past performance of low-level heuristics by applying the Multi-Armed Bandit (MAB) and selects one with the best performance. Their work consists of 18 low-level heuristics such as Gaussian mutation and differential mutation.

We are unaware of existing work in debugging deep learning models to apply hyperheuristics. To the best of our knowledge, Plum contributes as the first technique of this kind. Nonetheless, Plum only scratches the surface of using hyperheuristics for testing and debugging deep learning models. It has not combined different low-level strategies to form a sequence of repair strategies to be applied to a faulty DL model.

## VI. Conclusion

In this paper, we have presented Plum, the first and novel hyperheuristic approach to fixing deep learning models. Plum consists of two phases. The first phase is an exploration that takes a set of low-level repair strategies as inputs and generates a set of fixed model candidates. The second phase evaluates each low-level repair strategy by measuring the increases in validation and test accuracies of model candidates, followed by prioritizing these repair strategies by a prioritization scheme that balances among various metrics on the validation and test datasets. Finally, Plum generates a fixed DL model by applying the top-ranked repair strategy. The paper has also presented a novel low-level repair strategy to show the flexibility of Plum in incorporating new repair strategies. The experiment with five DL models has shown that Plum has improved test accuracy over the baseline by 2.49% and 3.11% on the CIFAR-10 and CIFAR100 datasets, outperforming two prior state-of-the-art deep learning debugging techniques in terms of test accuracy.

## References


[1] Y. LeCun, Y. Bengio, G. Hinton, "Deep learning," *Nature*, vol.521, pp.436-444, 2015.

[2] S. Martínez-Fernández, J. Bogner, X. Franch, M. Oriol, J. Siebert, A. Trendowicz, A. Vollmer, and S. Wagner, "Software Engineering for AI-Based Systems: A Survey," arXiv: 2015.01984, 2021.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkorekit, L. Jones, A. Gomez, and L.Kaiser, "Attention Is All You Need," In Proc. Advances in Neural Information Processing Systems, 2017, pp. 5998-6008.

[4] M. Tam and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," In Proc. 36th International Conference on Machine Learning, 2019, pp. 6105-6114.

[5] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness May Be at Odds with Accuracy," In Proc. International Conference on Learning Representations (ICLR), 2019.

[6] 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.

[7] S. Ma, Y. Liu, W. C. Lee, X. Zhang, A. Grama, "MODE: automated neural network model debugging via state differential analysis and input selection," In Proc. 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp.175-186.

[8] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "RobOT: Robustness-Oriented Testing for Deep Learning Systems," In Proc. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 300-311.

[9] H. Zhang and W.K. Chan, "Apricot: a weight-adaptation approach to fixing deep learning models," In Proc. 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 376-387.

[10] C. Shorten and T. Khoshgoftaar, "A survey on Image Data Augmentation for Deep Learning," Journal of Big Data, vol. 6(1), pp. 1-48, 2019.

[11] H. F. Eniser, S. Gerasimou, A. Sen, "DeepFault: Fault Localization for Deep Neural Networks," In Proc. International Conference on Fundamental Approaches to Software Engineering, 2019, pp. 171-191.

[12] B. Zoph, Q. Le, "Neural Architecture Search with Reinforcement Learning," arXiv: 1611.01578, 2016.

[13] K. He, X. Zhang, S. Ren, J. Sun, "Deep residual learning for image recognition," In Proc. IEEE conference on computer vision and pattern recognition, 2016, pp.770-778.

[14] A. G. Howard, M. Zhu, B. Chen, et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," arXiv:1704.04861, Apr 2017.

[15] G. Huang, Z. Liu, L. Maaten, K. Q. Weinberger, "Densely Connected Convolutional Networks," arXiv: 1608.06993, Jan 2018.

[16] S. Young, D. Rose, T. Karnowski, S. Lim, and R. Patton, "Optimizing Deep Learning Hyperparameters through an Evolutionary Algorithm," In Proc. Workshop on Machine Learning in High-Performance Computing Environments, pp. 1-5, 2015.

[17] PyTorch Training a Classifier, https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html.

[18] J. Goodfellow, J. Shlens, C. Szegedy, "Explaining and Harnessing Adversarial Examples," arXiv:1412.6572v3 [cs], Mar. 2015.

[19] S. Gu, L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," arXiv: 1412.5068 [cs], Dec. 2014.

[20] Keras: the Python deep learning API, https://keras.io/.

[21] TensorFlow: An end-to-end open source machine learning platform [Online]. Available: https://www.tensorflow.org/.

[22] A. Krizhevsky, G. Hinton, "Learning multiple layers of features from tiny images," Technical report, University of Toronto, 2009.

[23] M. Sandler, A. Howard, M. Zhu, et al. "MobileNetV2: Inverted Residuals and Linear Bottlenecks," arXiv: 1801.04381, Mar 2019.

[24] Convolutional Neural Networks for CIFAR-10, https://github.com/BIGBALLON/cifar-10-cnn.

[25] Keras Applications, https://keras.io/api/applications/.

[26] R. Khan, X. Zhang, R. Kumar, and E. Aboagye, "Evaluating the Performance of ResNet Model Based on Image Recognition," In Proc. 2018 International Conference on Computing and Artificial Intelligence, pp. 86-90, 2018.

[27] W. Wang, Y. Li, T. Zou, X. Wang, J. You, and Y. Luo, "A Novel Image Classification Approach via Dense-MobileNet Models," Mobile Information Systems, vol. 2020, article 7602384, 8 pages, 2020.

[28] Accuracy and Loss: Things to Know about the Top 1 and Top 5 Accuracy, https://towardsdatascience.com/accuracy-and-loss-things-to-know-about-the-top-1-and-top-5-accuracy-1d6beb8f6df3.

[29] PyTorch-cifar100, https://github.com/weiaicunzai/pytorch-cifar100

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

[31] Keras Optimizers, https://keras.io/api/optimizers/.

[32] PyTorch Examples, https://github.com/pytorch/examples.

[33] Trains a Resnet on the CIFAR10 dataset [Online]. Available: https://keras.io/examples/cifar10_resnet.

[34] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. "Gradient-based learning applied to document recognition." In Proceedings of the IEEE, 86(11):2278-2324, November 1998.

[35] T. Byun, V. Sharma, A. Vijayakumar, S. Rayadurgam, and D. Cofer, "Input Prioritization for Testing Neural Networks," In Proc. 2019 IEEE International Conference On Artificial Intelligence Testing (AITest), pp. 63-70, 2019.

[36] 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 Proc. 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), pp. 177-188, 2020.

[37] X. Zhang, J. Zhai, S. Ma, and C. Shen, "AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System," In Proc. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 359-371, 2021.

[38] M. Hassan, N. Sabar, and A. Song, "Optimising Deep Learning by Hyper-heuristic Approach for Classifying Good Quality Images," In Proc. International Conference on Computational Science, pp. 528-539, 2018.

[39] F. Ahmadizar, K. Soltanian, F. AkhlaghianTab, L. Tsoulos, "Artificial neural network development by means of a novel combination of grammatical evolution and genetic algorithm," Engineering Applications of Artificial Intelligence, vol. 39, pp.1-13, 2015.

[40] Backpropagation [Online]. Available: https://ml-cheatsheet.readthedocs.io/en/latest/backpropagation.html.

[41] B. Baker, O. Gupta, N. Naik, R. Rakar, "Designing Neural Network Architectures using Reinforcement Learning," arXiv: 1611.02167, Mar. 2017.

[42] E. Cantú-Paz, C. Kamath, "An empirical comparison of combinations of evolutionary algorithms and neural networks for classification problems," IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol.35, no.5, pp.915-927, 2005.

[43] N. Carlini, D. Wagner, "Towards Evaluating the Robustness of Neural Networks," In Proc. IEEE Symposium on Security and Privacy (SP), 2017, pp. 39-57.

[44] J. Chang, J. Sha, "Prune Deep Neural Networks With the Modified $L_{1/2}$ Penalty," IEEE Access, vol. 7, pp.2273-2280, 2019.

[45] A. Daniely, R. Frostig, Y. Singer, "Toward deeper understanding of neural networks: The power of initialization and a dual view on expressivity," Advances in Neural Information Processing Systems (NIPS), 2016, pp. 2253-2261.

[46] K. Deb, A. Anand, D. Joshi, "A computationally efficient evolutionary algorithm for real-parameter optimization," Evolutionary computation, vol.10, no.4, pp.371-395, 2002.

[47] J. Deng, W. Dong, R. Socher, L. Li, K. Li, L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," In Proc. IEEE conference on computer vision and pattern recognition, 2009, pp.248-255.

[48] J. Duchi, E. Hazan, Y. Singer, "Adaptive subgradient methods for online learning, and stochastic optimization," Journal of Machine Learning Research, Vol. 12, pp.2121-2159, 2011.

[49] C. L. Giles, G. M. Kuhn, and R. J. Williams, "Dynamic recurrent neural networks: Theory and applications," IEEE Trans. Neural Networks, vol.5, pp. 153–156, 1994.

[50] X. Glorot, Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," In Proc. the thirteenth international conference on artificial intelligence and statistics, 2010, pp. 249-256.

[51] I. Goodfellow, J. Pouget-Abadie, M. Mirza, et al., "Generative Adversarial Nets," In Advances in neural information processing systems, 2014, pp. 2672-2680.

[52] S. Han, J. Pool, S. Narang, et al., "DSD: regularizing deep neural networks with dense-sparse-dense training flow," arXiv: 1607.04381 [cs], 2016.

[53] J. Kim, R. Feldt, S. Yoo, "Guiding Deep Learning Systems Testing Using Surprise Adequacy," In Proc. 41st International Conference on Software Engineering (ICSE'19), 2019, pp. 1039-1049.

[54] D. Maclaurin, D. Duvenaud, R. P. Adams, "Gradient-based Hyperparameter Optimization through Reversible Learning," In Proc. International Conference on Machine Learning (ICML), 2015, pp. 2113-2122.

[55] MODE NN Debugging [Online]. Available: https://github.com/fabriceyhc/mode_nn_debugging [Accessed: Apr. 27, 2019].

[56] I. Szegedy, W. Zaremba, I. Sutskever. J. Bruna, D. Erhan, I. Goodfellow, R. Fergus, "Intriguing properties of neural networks," arXiv: 1312.6199v4 [cs], Feb 2014.

[57] I. Xiao, B. Li, J. Y. Zhu, W. He, M. Liu, D. Song, "Generating adversarial examples with adversarial networks," arXiv: 1801.02610 [cs], Feb. 2019.