2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-6654-3786-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SANER53432.2022.00128

NEURECOVER: Regression-Controlled Repair of Deep Neural Networks with Training History

Shogo Tokui
Fujitsu Limited
Kawasaki, Japan
tokui.shogo@fujitsu.com

Susumu Tokumoto
Fujitsu Limited
Kawasaki, Japan
tokumoto.susumu@fujitsu.com

Akihito Yoshii
Fujitsu Limited
Kawasaki, Japan
yoshii.akihito@fujitsu.com

Fuyuki Ishikawa
National Institute of Informatics
Tokyo, Japan
f-ishikawa@nii.ac.jp

Takao Nakagawa
Fujitsu Limited
Kawasaki, Japan
nakagawa-takao@fujitsu.com

Kazuki Munakata
Fujitsu Limited
Kawasaki, Japan
munakata.kazuki@fujitsu.com

Shinji Kikuchi
Fujitsu Limited
Kawasaki, Japan
skikuchi@fujitsu.com

***Abstract*—Systematic techniques to improve quality of deep neural networks (DNNs) are critical given the increasing demand for practical applications including safety-critical ones. The key challenge comes from the little controllability in updating DNNs. Retraining to fix some behavior often has a destructive impact on other behavior, causing regressions, i.e., the updated DNN fails with inputs correctly handled by the original one. This problem is crucial when engineers are required to investigate failures in intensive assurance activities for safety or trust.**

Search-based repair techniques for DNNs have potentials to tackle this challenge by enabling localized updates only on "responsible parameters" inside the DNN. However, the potentials have not been explored to realize sufficient controllability to suppress regressions in DNN repair tasks.

In this paper, we propose a novel DNN repair method that makes use of the training history for judging which DNN parameters should be changed or not to suppress regressions. We implemented the method into a tool called NEURECOVER and evaluated it with three datasets. Our method outperformed the existing method by achieving often less than a quarter, even a tenth in some cases, number of regressions. Our method is especially effective when the repair requirements are tight to fix specific failure types. In such cases, our method showed stably low rates (<2%) of regressions, which were in many cases a tenth of regressions caused by retraining.

***Keywords*-Deep Neural Network, Automated Program Repair, Fault Localization**

I. INTRODUCTION

Deep neural networks (DNNs) have recently been used in systems for applications such as speech recognition [1], machine translation [2], object detection [3], sentiment analysis [4], and face recognition [5]. They have also been used in safety-critical industrial applications such as medical diagnosis [6], autonomous driving [7], and aircraft collision avoidance systems [8]. However, engineering methodologies for the development, quality assurance, and operation of DNN-based systems were first discussed only a few years ago [9]–[11], and there are serious concerns about quality and continuous maintenance and improvement, in particular.

DNNs and other machine-learning-based software are referred to as Software 2.0 [12], which means software consists of enormous number of interconnected parameters and the behavior is derived in a data-driven way via training. This characteristic introduces a challenge in continuous improvement. Specifically, updates by retraining affect the whole behavior of the DNN. In other words, we do not have control to localize the changes to have limited impact on the specific behavior. This nature is even said as "Changing Anything Changes Everything" [13].

As DNNs have been applied to more safety-critical or quality-sensitive domains, suppressing regressions is increasingly crucial. For example, stakeholders are curious about whether there are unacceptable mistakes with high risks that can lead to serious hazards or distrust by stakeholders. In such a case, engineers are required to have costly activities to check failed cases and give some explanation. Regressions lead to high cost of redoing such activities. This is true even if the total accuracy is improved as the combined effect of improvements and regressions.

Traditional software engineering techniques have great potentials to tackle problems in DNNs. Techniques for automated program repair have potentials to realize effective methods for automated DNN repair. Many techniques have been proposed to fix programs, especially the "Generate and Validate" technique, which has evolved significantly in the last decade and has been highly successful in fixing simple faults [14]–[16].

There has already been a study to apply the search-based approach for the DNN repair problem. Sohn et al. proposed Arachne [17], a method for turning misclassified data into correctly classified data by changing the parameters (weights) of the DNN model in an exploratory manner. This method consists of fault localization to identify the weights causing misclassification and particle swarm optimization [18], [19] to find weight values that will reduce the error. However, because the fault localization of Arachne identifies the target weights by considering only their impact on the misclassified data, there is a high possibility that the method will turn correctly classified data into misclassified data. Arachne thus shares the common problem of regressions as retraining.

978-1-6654-3786-8/22/$31.00 ©2022 IEEE
DOI 10.1109/SANER53432.2022.00128

In this paper, we propose a novel DNN repair technique, NEURECOVER, that suppresses regressions by using the training history in the fault localization step. The basic idea of NEURECOVER is to find the point in the training history when the model correctly classified a certain data sample that is now misclassified, and then to identify weights that can safely correct the misclassification, by comparing the past model with the current model. Specifically, NEURECOVER identifies weights with the following properties: their values have changed significantly in the training process, and they do not affect the output for improved data (i.e., data that was first misclassified but then classified correctly in the training process), but they do affect the output for regressed data (i.e., data that was once correctly classified but then misclassified in the training process). Then, by applying particle swarm optimization on the identified weights, NEURECOVER can update the DNN model to obtain more improved data and less regressed data.

We experimentally evaluated NEURECOVER with models with three DNN architectures by using three image classification datasets, GTSRB, CIFAR-10, and Fasihon-MNIST. NEURECOVER outperformed the baseline method Arachne by achieving often less than a quarter, even a tenth in some cases, number of regressions. NEURECOVER is especially suitable when the repair requirements are tight to fix specific failure types while avoiding regressions. In such cases, NEURECOVER showed stably low rates ($<2\%$) of regressions, in many cases a tenth, at most a quarter, compared with retraining that tend to have large shuffling of success and failure cases.

The contributions of this paper are summarized as follows:

- A novel DNN repair technique that suppresses regression by using the training history.
- An implementation of the technique, called NEURECOVER, including algorithm improvements from Arachne, an existing method for search-based DNN repair.
- Experiments to investigate repair performance in the design space for search-based DNN repair methods.
- Experiments to investigate repair performance with loose and tight requirements, respectively, for search-based DNN repair methods as well as common retraining.

The remainder of this paper is structured as follows. Section II describes DNNs and the existing technique, Arachne, as the background of this study. Section III describes the proposed technique, NEURECOVER. Section IV describes the evaluation experiment and discusses the proposed technique and its validity. Finally, section V describes related works, and section VI summarizes this paper and our future works.

II. BACKGROUND

This section describes DNNs as well as Arachne, an existing DNN repair technique that directly corrects the parameters (weights) of DNN models without retraining.

A. Deep Neural Network (DNN)

A DNN is a neural network composed of an input layer, an output layer, and two or more hidden layers. In particular, a feedforward neural network (FFNN) is known as a primary neural network to solve classification problems. An FFNN, propagates information through an input layer, a hidden layer, and an output layer, in order, and it outputs a prediction label for the input.

This section explains how to train a DNN model. The model has the DNN architecture and parameter values for the hidden layer. Each parameter value of the hidden layer is adjusted using training data. For data x given by the input layer, the hidden layer converts it to $o = wx + b$ via two parameters, a weight w and a bias b; then, $x' = A(o)$ is outputted via a nonlinear derivative function A called an activation function. The output layer obtains the index of the largest element of the hidden layer's output and gives its prediction result. The function representing the error between the expected and predicted labels for the data is called the loss function L. For instance, the squared error is one such loss function. A smaller loss indicates a better model. In the training of a DNN model, the parameters are adjusted by using the error backpropagation method to reduce the loss [20]. The error backpropagation method executes the steepest descent method, in which it adjusts the weights to $w = w - \alpha\frac{\partial L}{\partial w}$ by using the learning rate α. When a model is trained for n epochs, the error backpropagation method is repeated n times.

The field of image recognition uses convolutional neural networks (CNNs). A CNN is a DNN in which a convolutional layer for image processing is added to the hidden layer. Whereas the convolutional layer propagates to the subsequent stage via one feature, which convolves part of the region of the neurons in the previous stage, the layer to which all the neurons in the previous and subsequent stages are connected is called the fully connected layer. A basic CNN thus consists of four layers: an input layer, a convolutional layer, a fully connected layer, and an output layer.

A system using a DNN can build a model by training with data. When erroneous behavior is detected during operation or testing of the system, the DNN model is modified by adding data and retraining. However, retraining requires additional data to correct misclassified data, and it is not always possible to correct a model's output with added data. Therefore, to correct a DNN model without retraining, techniques have been studied to correct misclassification by directly manipulating the values of the model's weight parameters.

B. Search-Based DNN Repair

Arachne is a proposed DNN repair technique that locally modifies a DNN without retraining by changing the model's parameters (i.e., the weights) in an exploratory manner [17]. It works by identifying a weight that induces misclassification in the trained model, adjusting the value of the weight by using particle swarm optimization [18], [19], and correcting the misclassification to the expected classification.

Arachne's repair process consists of two steps: fault localization and particle swarm optimization. In fault localization, the gradient of each weight loss function in the DNN model and the output value of each layer are calculated as the impacts of the weights on misclassified data, and the weight causing misclassification is identified. In particle swarm optimization, by applying a fitness function, the weight value specified by fault localization is optimized to increase the amount of misclassified data that can be correctly classified. The details of fault localization and particle swarm optimization are described below.

1) Fault Localization: Because a DNN model includes more than tens of thousands of weights, it is very expensive to adjust all weights at the same time by particle swarm optimization. Accordingly, to narrow down the weights to be optimized, in the fault localization step Arachne focuses on weights connected to the final layer and then tries to identify those that have a large impact on misclassification. It uses two methods to evaluate each weight's impact on a specific misclassification: (1) the **gradient of the loss function**, which is used to adjust the weight and bias, and (2) the **output value of forward propagation**, which indicates the activation of neurons during model training.

First, Arachne inputs a misclassified sample to the DNN model to be repaired. Next, it obtains the values of the weight parameters in the final layer. Finally, it ranks the weights by considering both the gradient of the loss function and the output value of forward propagation. The gradient of the loss function is the value $\frac{\partial L}{\partial w}$ obtained by differentiating the loss function L by the weight w; it is calculated as $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial o}\frac{\partial o}{\partial w}$ by using the output o of forward propagation in the final layer. If the output of the j-th neuron in the last layer is o_j, then the gradient of the loss function with respect to the weight $w_{i,j}$ and the i-th neuron in the previous layer is calculated as $\frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial o_j}\frac{\partial o_j}{\partial w_{i,j}}$. The output value of forward propagation for weight $w_{i,j}$ is calculated by multiplying the output o_i of the layer before the activation function's nonlinear conversion by the weight $w_{i,j}$, i.e., as $o_i \cdot w_{i,j}$.

The number of candidate weights selected according to the gradient of the loss function is determined to be N_g in advance. That is, the weights are sorted by their gradients, and the top N_g weights are treated as candidates. Finally, to extract the set of weights to be optimized in the next step, the Pareto front is calculated after performing multi-objective optimization with both the gradient of the loss function and the output values of forward propagation as objective functions.

2) Patch Generation: Arachne corrects a DNN model's misclassification by using particle swarm optimization for the weights specified by fault localization [18], [19]. Particle swarm optimization is known to be effective for optimization in a continuous space and is suitable for unrestricted weight modification in the range of real numbers.

Arachne expresses the particle positions in particle swarm optimization, with the set of weights specified by fault localization, as a vector $\vec{x}$. The current particle vector $\vec{x}_t$ is updated using the velocity vector $\vec{v}_{t+1}$ via Equation 1 below. The current velocity vector $\vec{v}_t$ is updated via Equation 2 by using the current particle vector $\vec{x}_t$; a particle vector $\vec{p}_l$, which takes the best fit value among the particle's previously observed values; a particle vector $\vec{p}_g$, which takes the best fit value for the whole group; and a uniform random number $U(\phi)$ $(0 \leq U(\phi) \leq \phi)$. Here, ϕ_1 and ϕ_2 control the convergence of particles in a group without setting explicit velocity boundaries for local and global components, respectively. The value χ, which is called a constriction factor, is calculated from ϕ_1 and ϕ_2 via Equation 3. Arachne uses the same values for ϕ_1 and ϕ_2. It extracts the particle vector's initial value $\vec{x}_0$ from a normal distribution determined by the distribution of weights, with the initial velocity $\vec{v}_0$ set to $\vec{0}$.

$$\vec{x}_{t+1} \leftarrow \vec{x}_t + \vec{v}_{t+1} \tag{1}$$

$$\vec{v}_{t+1} \leftarrow \chi(\vec{v}_t + U(\phi_1)(\vec{p}_l - \vec{x}_t) + U(\phi_2)(\vec{p}_g - \vec{x}_t)) \tag{2}$$

$$\chi \leftarrow \frac{2}{\phi - 2 + \sqrt{\phi^2 - 4\phi}}, \text{ where } \phi = \phi_1 = \phi_2 \tag{3}$$

According to the fitness function given in Equation 4 below, $\vec{p}_l$ and $\vec{p}_g$ in Equation 2 use the particles with the best fitness values among the particles observed in the past. Here, I_{neg} is a set of misclassified samples, and I_{pos} is a set of randomly selected samples that were correctly classified. Lastly, N_{patched} is the number of data instances in I_{neg} that were changed from misclassification to the expected classification, whereas N_{intact} is the number of data instances in I_{pos} that were not changed from the expected classification to misclassification.

$$\textit{fitness} = \frac{N_{\text{patched}} + 1}{L(I_{\text{neg}}) + 1} + \frac{N_{\text{intact}} + 1}{L(I_{\text{pos}}) + 1} \tag{4}$$

III. NeuRecover: DNN Repair with Training History

Arachne identifies the parameters (weights) that affect misclassification and searches for weight values that reduce the error by using particle swarm optimization. However, Arachne has a potential risk of introducing new misclassification into the model while correcting some misclassification. We consider the cause of this problem to be that Arachne uses only misclassified data as information for fault localization and does not consider correctly classified data. In other words, if some of the weights identified by the fault localization step affect data that was correctly classified, then the patch generation step may turn correctly classified samples into misclassified samples. To solve this problem, we made the following two assumptions: (1) It should be relatively easy to correct misclassification if a misclassified sample was once classified correctly during the training process. (2) If we identify weights that impact the results for correctly classified samples, then we can reduce data regression by avoiding changes to the values of those weights.

Hence, we propose NeuRecover, which is a novel DNN repair technique that uses fault localization with the training history. In this technique, the fault localization detects

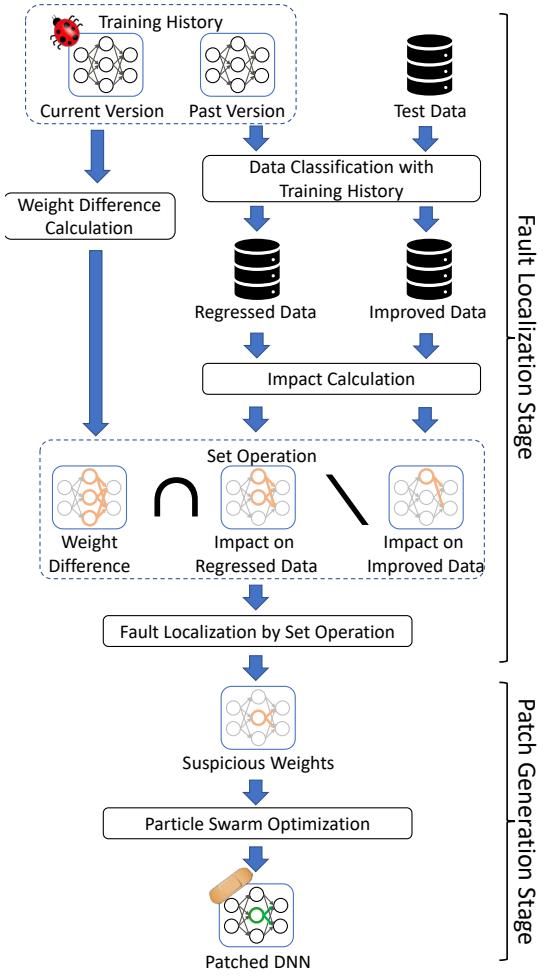


Fig. 1. Overview of NEURECOVER

improved data and regressed data in the training process. Here, we use the term *improved data* to refer to data that changed from misclassified to correctly classified and the term *regressed data* to refer to data that changed from correctly classified to misclassified. Then, as illustrated in Figure 1, NEURECOVER identifies weights that have changed significantly during the training process and do not affect improved data but only affect regressed data (Fault Localization Stage), and optimizes the localized weights by particle swarm optimization as in Arachne (Patch Generation Stage). The fault localization step using the training history is executed in the following three steps.

Step i Data classification with training history
Step ii Impact calculation
Step iii Fault localization by set operation

In Step i, NEURECOVER uses the training history to detect regressed and improved data in the test data. In Step ii, it calculates the impact on the regressed data, the impact on the improved data, and the difference between the weights. In Step iii, it identifies the sets of weights affecting the regressed data, the improved data, and the difference between the weights, and it uses a set operation to localize these weights. Finally, in patch generation stage, NEURECOVER corrects the localized weights of the DNN model by using particle swarm optimization. We describe the details of each step below.

A. Step i: Data Classification with Training History

First, NEURECOVER detects regressed data and improved data from the training history. To repair of a model that was trained for n epochs, it uses the weights of models M_n and M_{n-k} in the training history, where M_i denotes the model that was trained for $i (1 \leq i \leq n)$ epochs.

For each model, NEURECOVER examines the predictions of the dataset. It classifies data that changed from the expected classification by M_{n-k} to misclassification by M_n as regressed data, and data that changed from misclassification by M_{n-k} to the expected classification by M_n as improved data. In our evaluation experiment described in this paper, the datasets were classified using the models M_{n-1} and M_n $(k = 1)$.

B. Step ii: Impact Calculation

Next, NEURECOVER calculates five impacts to identify the weights to be corrected: the weight difference, w_{diff}; the backward impact on regressed data, $back_{\text{reg}}$; the forward impact on regressed data, fwd_{reg}; the backward impact on improved data, $back_{\text{imp}}$; and the forward impact on improved data, fwd_{imp}. In this study, the backward impact is given by the gradient of the loss function, and the forward impact is given by the output value of forward propagation.

We assume that a large weight difference w_{diff} is the cause of changes in the prediction results. NEURECOVER thus obtain a weight array w_n from M_n and a weight array w_{n-k} from M_{n-k} and calculates $w_{\text{diff}} = w_n - w_{n-k}$.

The backward impact *back* is calculated as $back = \frac{\partial L}{\partial w_{i,j}} = \frac{\partial L}{\partial o_j}\frac{\partial o_j}{\partial w_{i,j}}$ for the localized weights $w_{i,j}$, which connect the j-th neuron of the previous layer and the i-th neuron of the target layer, and the output activation value o_j of the j-th neuron of the target layer. *back* thus depends on the loss function L, the neuron output o, the weights w, and the inputs. NEURECOVER obtains $back_{\text{reg}}$ as the backward impact on regressed data and $back_{\text{imp}}$ as the backward impact on improved data.

The forward impact *fwd* is calculated as $fwd = o_i \cdot w_{i,j}$ from the output activation value o_i of the i-th neuron of the previous layer and the weights $w_{i,j}$. It thus depends on the weights w and the inputs. NEURECOVER obtains fwd_{reg} as the forward impact on regressed data and fwd_{imp} as the forward impact on improved data.

Note that NEURECOVER localizes the weights of all fully connected layers, not just the last layer of the DNN model. That is, it computes the backward and forward impact for each layer of the DNN model. In this paper, for hypothesis verification in the initial stage, only the fully connected layers are examined, but in the future, we will consider correcting the weights of the convolutional layers, as well.

C. Step iii: Fault Localization by Set Operation

In the last step, the weights are sorted for each of the five impacts w_{diff}, $back_{\text{reg}}$, fwd_{reg}, $back_{\text{imp}}$, and $back_{\text{imp}}$. Then, the five corresponding sets W_{diff}, B_{reg}, F_{reg}, B_{imp}, and F_{imp} of the top N_g weights are obtained. Finally, the weights specified by the set operation in Equation 5 are defined as the target of DNN model repair.

$$W_{\text{localized}} = (B_{\text{reg}} \cap F_{\text{reg}}) \cap W_{\text{diff}} \backslash (B_{\text{imp}} \cap F_{\text{imp}}) \tag{5}$$

By Equation 5, NEURECOVER identifies a localized set $W_{\text{localized}}$ of weights that have a large difference and affect the regressed data, while excluding weights that affect the improved data. We consider weights that affect the regressed data and improved data to have large values for both the backward and forward impacts. Therefore, the weights that affect the regressed data are given by $B_{\text{reg}} \cap F_{\text{reg}}$, and the weights that affect the improved data are given by $B_{\text{imp}} \cap F_{\text{imp}}$. The set operation in Equation 5 thus suppresses data regression during fault localization and repair of the DNN model.

D. Patch Generation Stage

In the patch generation stage, NEURECOVER corrects the localized weights by using particle swarm optimization as described in section II-B2. For this paper, however, we changed the fitness function and the samples chosen from the data that was correctly classified.

The fitness function used in Arachne is positively proportional to the number of corrected data instances and intact data instances. This means that the fitness value depends on the number of misclassified data instances and sampled correctly classified data instances, and is considered to be oversensitive to the absolute amount of these data instances. To mitigate the sensitivity and obtain stable results, we changed the fitness function to use a relative amount of misclassified and correctly classified data before and after running the method. α is a hyper-parameter to adjust the degree of regression suppression.

$$\textit{fitness} = \frac{N_{\text{patched}}/|I_{\text{neg}}| + 1}{L(I_{\text{neg}}) + 1} + \alpha \cdot \frac{N_{\text{intact}}/|I_{\text{pos}}| + 1}{L(I_{\text{pos}}) + 1} \tag{6}$$

Note also that Arachne randomly selects the samples of correctly classified data. However, the larger the mean square error between the predicted and correct values for the correctly classified data is, the closer the that data is to the classification boundaries. We consider prevention of the regression of correctly classified data that is close to the classification boundaries to also prevent regression of other correctly classified data. Accordingly, NEURECOVER selects the samples of correctly classified data in order of the error size.

IV. EVALUATION

In this section, we describe an evaluation experiment of the proposed technique NEURECOVER.

In the experiment, we compared three methods, NEURECOVER, Arachne, and retraining, and evaluated the design validity of NEURECOVER on three datasets and three model architectures for image classification.

A. Experiment Setup

1) Model Architectures and Datasets: To avoid biasing the evaluation toward any particular model, we tried nine combinations of model architectures and datasets. We prepared three different model architectures: 8-layer CNN (8CN), VGG16 (V16), and VGG19 (V19). The 8-layer CNN consists of six convolutional layers and two fully connected layers, the VGG16 consists of 13 convolutional layers and three fully connected layers, and the VGG19 consists of 16 convolutional layers and two fully connected layer. We also prepared three different image classification datasets: GTSRB (GT) [21], CIFAR-10 (C10) [22] and Fashion-MNIST (FM) [23].

2) Data Split and Categorization: Each dataset was divided into three categories: *train*, *repair*, and *test*. A *repair* category was specially defined, being separated from a *train* category for the debugging process.

The *train* category is used at a training prior to the fault localization steps described in the section III. During the fault localization steps, the data classification proceeds with data samples taken from the *repair* category. After the optimization process has been completed, a repaired model is evaluated using the *test* category samples.

Since the datasets are originally split into two categories, train and test, we split the original train data into two new categories, *train* and *repair*, as in K-fold cross validation. Multiple patterns of a separation between *train* and *repair* can be defined. Let $K \in \mathbb{N}$ as the number of patterns. The whole part of the original train data can be divided into K segments. We define a *repair* category as one of the K segments and define a *train* category as data samples basically including $K-1$ segments; therefore, K possibilities of combinations of the *repair* and the *train* category can be considered.

Models trained only on the *train* category data are regarded as faulty baseline models, and the models are subject to correction in each technique with the *repair* category data. The data classification results in the baseline models with the *test* category data are shown in Table I. We have chosen the $K = 5$ condition.[1] The #pos, #neg, #reg, and #imp are the mean number of correctly classified, misclassified, regressed, and improved data for $K = 5$ patterns, respectively. The #pos and the #neg are calculated from the classification results by a model trained until the last epoch (i.e. the 10th epoch). On the other hand, the #reg and the #imp show the change of the classification results at the last epoch in comparison of the one before epoch (i.e. the 9th epoch).

3) Competitors: The experiment compared our proposed method with Arachne and retraining. We implemented the experimental code for Arachne according to the Arachne paper because its implementation was not published. Retraining is a method that attempts to improve the model by adding data that

[1]We experimented with 4 patterns of the 5 combinations for $K = 5$ segments due to a defect in the experiment source code and time constraints.

TABLE I
THE CLASSIFICATION RESULTS OF BASELINE MODELS

Datasets	Model Arch.	Epochs	ACC	#pos	#neg	#reg	#imp
GTSRB	8-layer CNN	10	96.247	12156.0	474.0	194.3	166.8
GTSRB	VGG16	10	89.733	11333.3	1296.8	319.3	344.0
GTSRB	VGG19	10	53.830	6798.8	5831.3	399.0	441.3
CIFAR10	8-layer CNN	10	74.580	7458.0	2542.0	734.3	906.8
CIFAR10	VGG16	10	80.880	8088.0	1912.0	487.5	477.0
CIFAR10	VGG19	10	57.738	5773.8	4226.3	326.0	350.5
Fashion-MNIST	8-layer CNN	10	90.555	9055.5	944.5	256.5	251.3
Fashion-MNIST	VGG16	10	91.110	9111.0	889.0	153.0	170.5
Fashion-MNIST	VGG19	10	83.850	8385.0	1615.0	179.8	194.5

is not included in the training dataset and training again with that data. Developers and maintainers of ML systems generally use it for repairing their ML model when they find faults in the model. In the experiment, we call "retraining" the same epochs training as the baseline model from the initial state with the *train* category data, misclassified data in *repair* category, and sampled correctly classified data in the *repair* category. The "retraining" allows us to observe only the effects of the increased *repair* category data in training. Note that since the baseline model has not been trained for a sufficient number of epochs, additional training of the models will increase the accuracy with or without the *repair* category data.

4) Metrics: The DNN repair performance was evaluated in terms of the accuracy, repair rate, and break rate, which are given by the following equations.

$$\textit{Accuracy (ACC)} = |I_{\text{pos}}|/|I_{\text{all}}| \times 100$$
$$\textit{Repair Rate (RR)} = |I_{\text{imp}}|/|I_{\text{neg}}| \times 100$$
$$\textit{Break Rate (BR)} = |I_{\text{reg}}|/|I_{\text{pos}}| \times 100$$

Here, ACC, which is the percentage of correctly classified data in all the test data, indicates the model's performance. ΔACC is the difference of the ACC values between the repaired model and the original model. RR is the ratio of correctly classified data among the data misclassified by the original model before repair. BR is the ratio of misclassified data among the data correctly classified by the original model.

It should be noted the impact of BR values is larger than that of RR as generally $|I_{\text{pos}}|$ is much larger than $|I_{\text{neg}}|$. For example, breaking 1% of positive inputs and repairing 1% of negative inputs mean the impact of regressions is very dominant. We also look at the number of broken and repaired (patched) samples to investigate the trade-off. In addition, we expect suppressing regressions, i.e., achieving stably low RR values, is a unique and effective feature of NEURECOVER.

5) Experimental Environment: NEURECOVER and Arachne were implemented in Python 3.6.9, and the DNN training models were implemented in TensorFlow 2.4.1.

NEURECOVER has several hyper-parameters. PSO used $\phi_1 = \phi_2 = 4.1$, and the maximum number of iterations is 100. PSO uses a population size of 200. In addition, the weight rate of our fitness function α is 1. However, in the experiment for specific misclassified data, the weight rate α is 5.

B. Research Questions

We conducted the experiment to evaluate the effectiveness of NEURECOVER by answering the following research questions.

- RQ1 Are the design elements of NEURECOVER beneficial?
  - RQ1-1 How does use of the training history affect the repair performance?
  - RQ1-2 How do the other variations in the repair method affect the repair performance?
- RQ2 Is NEURECOVER effective in controlling regressions in repair tasks?
- RQ3 Is NEURECOVER effective in controlling regressions in fine-grained repair tasks for specific failure types?

We investigate effectiveness of each technical feature in NEURECOVER compared with the baseline method Arachne in RQ1. RQ1-1 is about the core feature of NEURECOVER to make use of the training history and RQ1-2 covers the other algorithm improvements. RQ2 and RQ3 evaluate the repair performance of NEURECOVER, including all the features, with Arachne and retraining. We consider basic repair tasks and fine-grained repair tasks that do not or do focus on failure types, i.e., labels, respectively. We expect NEURECOVER is more effective in fine-grained repair tasks where we have tight requirements on what to repair and thus hints from the localization phase help avoid manipulating unnecessarily large number of weight values.

C. Results

1) **RQ1-1. How does use of the training history affect the repair performance?***:* The key idea of NEURECOVER is to make use of the training history in the localization phase. Specifically, we focused on weights whose values changed a lot and weights that affected improved data (Section III-C. In RQ 1-1, we investigate how these two points work.

The results are shown in Table II. The best values for each metric (ΔACC, RR, BR) are shown in bold. The rightmost column (*Without Improved Data*) has large RR values but sometimes also large BR values, resulting in less ACC values. This point suggests that the idea to avoid manipulating weights that contributed improvement in the training history is working as expected. The left and center columns have comparative scores ($AllImpact$ and $WithoutDiff$) but the left tends to

TABLE II
RQ1-1. COMPARISON ABOUT USE OF THE TRAINING HISTORY

Impact	All Impact			Without Diff			Without Improved Data		
Model	ΔACC	RR	BR	ΔACC	RR	BR	ΔACC	RR	BR
GT+8CN	-1.067	**13.418**	1.679	**0.067**	12.981	0.448	0.063	11.856	**0.420**
GT+V16	**0.374**	18.114	**1.714**	-1.083	**23.526**	3.975	-0.148	21.693	2.725
GT+V19	**-1.015**	8.476	**9.158**	-2.007	8.302	10.857	-3.702	**10.305**	15.716
C10+8CN	-2.105	5.976	4.978	**0.322**	**6.276**	**1.750**	-0.248	4.844	2.065
C10+V16	-0.090	16.727	4.081	**0.743**	16.121	**2.978**	-0.735	**21.202**	6.069
C10+V19	-1.420	8.629	8.777	**-1.033**	7.845	**7.530**	-2.548	**11.152**	12.573
FM+8CN	**-0.020**	6.161	**0.684**	-0.720	**10.589**	1.930	-0.343	9.208	1.353
FM+V16	**0.160**	15.074	**1.310**	-0.433	**23.128**	2.748	-0.213	19.717	2.168
FM+V19	**-0.083**	7.450	**1.536**	-1.435	12.407	4.100	-2.140	**14.124**	5.277

TABLE III
RQ1-2 (PARTIAL). COMPARISON OF FITNESS FUNCTIONS

Fitness	NEURECOVER			Arachne		
Model	ΔACC	RR	BR	ΔACC	RR	BR
GT+8CN	-1.067	**13.418**	1.679	**0.032**	8.779	**0.326**
GT+V16	**0.374**	**18.114**	**1.714**	0.329	17.549	1.726
GT+V19	**-1.015**	8.476	**9.158**	-5.780	**12.656**	21.600
C10+8CN	**-2.105**	5.976	**4.978**	-10.278	**17.947**	19.928
C10+V16	**-0.090**	16.727	**4.081**	-7.958	**37.529**	18.723
C10+V19	**-1.420**	8.629	**8.777**	-11.735	**19.592**	34.662
FM+8CN	**-0.020**	6.161	**0.684**	-5.570	**28.136**	9.080
FM+V16	**0.160**	15.074	**1.310**	-9.380	**39.973**	14.196
FM+V19	**-0.083**	7.450	**1.536**	-11.925	**30.590**	20.115

have low BR. We claim that the proposed $AllImpact$ is more stable when we are concerned about regressions.

Answer to RQ1-1

The proposed ideas to use the training history in NEURECOVER contribute to suppress regressions in DNN repair.

2) **RQ1-2. How do the other variations in the repair method affect the repair performance?***:* We had a few improvements in NEURECOVER compared with the baseline Arachne implementation. Specifically, we included many layers as the target of repair, changed the way of sampling to calculate the fitness to reflect the loss, and changed the fitness function to be relative to the number of samples. As RQ1-2, we experimentally confirmed these changes some or less contribute to the repair performance.

We omit the concrete results for the first two aspects due to space limitation. Table III shows the results on the third point, the fitness function, which had the largest impact on the repair performance. The modified fitness function contributes to suppress the regressions (low BR), resulting in better overall accuracy (high ΔACC).

Answer to RQ1-2

The algorithm improvements in NEURECOVER, especially in the fitness function, contribute to the DNN repair performance for suppressing regressions.

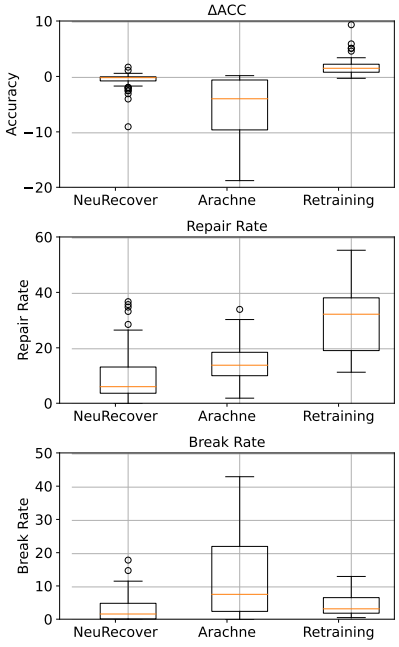


Fig. 2. Comparison of NEURECOVER, Arachne and Retraining (the increase rate of ACC, RR and BR)

3) **RQ2. Is NEURECOVER effective in controlling regressions in repair tasks?***:* We compared the repair performance of NEURECOVER, i.e., ACC, RR, and BR, with the baseline Arachne and also with Retrain. The total results are summarized in Figure 2 and the detailed comparisons with each target are shown in Tables IV and V, respectively.

We start with discussion on NEURECOVER and Arachne in in Table IV. NEURECOVER achieved better (lower) BR scores than Arachne. BR scores of Arachne are unstable and often very high (many over 10% and at worst even almost 40%). BR scores of NEURECOVER are stably low (below 10%). NEURECOVER often had less than a quarter, even a tenth in some cases, number of regressions compared with Arachne. As a result, ACC values are better in NEURECOVER in most cases. Arachne achieved better (higher) RR scores in

TABLE IV
RQ2. COMPARISON BETWEEN NEURECOVER AND ARACHNE

				NEURECOVER			Arachne		
Datasets	Model Arch.	Epochs	orig ACC	ACC	RR	BR	ACC	RR	BR
GTSRB	8-layer CNN	5	95.893	94.662	**11.856**	1.817	**94.751**	10.262	**1.649**
GTSRB	8-layer CNN	10	96.247	95.180	**13.418**	1.679	**95.907**	9.728	**0.732**
GTSRB	VGG16	5	87.201	**87.223**	**27.482**	4.064	86.158	15.949	**3.533**
GTSRB	VGG16	10	89.733	**90.107**	**18.114**	1.714	89.515	8.482	**1.205**
GTSRB	VGG19	5	51.653	**51.247**	5.476	**5.909**	43.452	**12.696**	27.744
GTSRB	VGG19	10	53.830	**52.815**	8.476	**9.158**	45.283	**13.062**	27.083
CIFAR10	8-layer CNN	5	67.635	**67.643**	0.200	**0.088**	61.045	**11.414**	15.018
CIFAR10	8-layer CNN	10	74.580	**72.475**	5.976	**4.978**	67.495	**13.025**	13.925
CIFAR10	VGG16	5	78.885	**79.158**	4.289	**0.805**	77.538	**15.495**	5.882
CIFAR10	VGG16	10	80.880	**80.790**	**16.727**	4.081	79.918	10.915	**3.803**
CIFAR10	VGG19	5	55.905	**55.030**	6.702	**6.830**	42.273	**19.359**	39.651
CIFAR10	VGG19	10	57.738	**56.318**	8.629	**8.777**	43.475	**18.411**	38.182
Fashion-MNIST	8-layer CNN	5	89.495	**89.500**	0.720	**0.078**	85.623	**14.730**	6.059
Fashion-MNIST	8-layer CNN	10	90.555	**90.535**	6.161	**0.684**	87.028	**16.494**	5.620
Fashion-MNIST	VGG16	5	89.923	**90.070**	12.225	**1.278**	87.368	**16.294**	4.715
Fashion-MNIST	VGG16	10	91.110	**91.270**	**15.074**	1.310	91.140	5.746	**0.524**
Fashion-MNIST	VGG19	5	82.453	**81.313**	8.973	**3.293**	70.335	**27.305**	20.515
Fashion-MNIST	VGG19	10	83.850	**83.768**	7.450	**1.536**	70.885	**26.252**	20.526

TABLE V
RQ2. COMPARISON BETWEEN NEURECOVER AND RETRAINING (PARTIAL)

				NEURECOVER			Retraining		
Datasets	Model Arch.	Epochs	orig ACC	ACC	RR	BR	ACC	RR	BR
GTSRB	8-layer CNN	10	96.247	95.180	13.418	1.679	**96.958**	**45.730**	**1.073**
GTSRB	VGG16	10	89.733	90.107	18.114	**1.714**	**91.059**	**29.905**	1.993
GTSRB	VGG19	10	53.830	52.815	8.476	9.158	**55.689**	**12.281**	**7.090**
CIFAR10	8-layer CNN	10	74.580	72.475	5.976	**4.978**	**76.032**	**38.147**	11.065
CIFAR10	VGG16	10	80.880	80.790	16.727	**4.081**	**83.715**	**36.172**	5.061
CIFAR10	VGG19	10	57.738	56.318	8.629	8.777	**59.787**	**13.759**	**6.521**
Fashion-MNIST	8-layer CNN	10	90.555	90.535	6.161	**0.684**	**91.060**	**35.220**	3.122
Fashion-MNIST	VGG16	10	91.110	91.270	15.074	**1.310**	**92.045**	**30.777**	1.978
Fashion-MNIST	VGG19	10	83.850	83.768	7.450	**1.536**	**85.172**	**19.720**	2.224

TABLE VI
RQ3. COMPARISON OF LABEL-WISE REPAIR BETWEEN NEURECOVER, ARACHNE AND RETRAINING

		NEURECOVER			Arachne			Retraining		
Model	LW-#neg	ΔACC	LW-RR	BR	ΔACC	LW-RR	BR	ΔACC	LW-RR	BR
C10+8CN	448.0	**-0.405**	12.444	**1.759**	-2.270	**37.612**	6.480	-0.480	35.938	12.672
C10+V16	339.3	0.333	4.643	**0.489**	-0.995	**33.161**	3.725	**1.227**	17.318	4.944
C10+V19	603.8	-0.155	5.880	**2.035**	-7.317	**49.358**	23.708	**0.082**	9.400	8.547
FM+8CN	292.8	-0.165	8.198	**0.546**	-1.383	19.129	2.888	**0.025**	**22.545**	3.131
FM+V16	243.3	**0.135**	1.953	**0.272**	-0.235	9.455	0.732	-0.313	**15.313**	2.814
FM+V19	414.8	-0.002	1.989	**0.200**	-9.188	**70.283**	15.729	**0.200**	9.222	2.289

most cases but the regressions negated the improvement. These points are also summarized in the box plot of Figure 2.

Comparison with retraining is shown in Table V. This table is partial only for Epochs=10 due to space limitation as the omitted parts had very similar tendency. In general, retraining shows better repair performance though NEURECOVER keeps better (lower) BR values. One hypothesis is that the potential of search-based repair is not in repairing any failed inputs but in repairing specific failed inputs. Retraining can take the freedom to pick up "easy-to-fix" failed inputs and sufficiently works. This point is investigated in the following RQ3.

Answer to RQ2

NEURECOVER outperforms the baseline, Arachne, by stably suppressing regressions. Retraining is appropriate when the repair requirements are not tight, i.e., when improvements for any failed inputs are appreciated and some regressions are accepted. NEURECOVER is a good option when the number of regressions is critical.

4) **RQ3. Is NEURECOVER effective in controlling regressions in fine-grained repair tasks for specific failure types?***:* We evaluated repair performance in terms of fine-grained

control. Specifically, we consider popular scenarios in which a specific type of failures occur too frequently and we want to repair it. We picked up models with epochs=10 and defined the repair target by investigating the model performance. For CIFAR-10, the repair target was set as misclassification of label 3 to 5 (cat to dog). For Fashion-MNIST, the target was misclassification of label 6 to 0 (shirts to T-shirts). Both are representatives of confusing (visually close) labels. GTSRB was not included as it has many labels and the number of data for each specific failure type is too small.

Table VI shows the results. The negative data (failed inputs) are considered for the specific label (in the label-wise way: LW). Thus, LW-#neg and LW-RR denote the number of negative (failed) inputs and the repair rate for the label, respectively. NEURECOVER outperforms Arachne with stably low BR (less than 2%). Retraining shows worse ACC and BR compared with the case of RQ2 (Table V). As the result, NEURECOVER showed good controllability with stably low BR, with in many cases a tenth, at most a quarter, number of regressions compared with retraining.

Figure 3 shows detailed label-wise repair performance. We picked up results for VGG16 with CIFAR-10 and Fashion-MNIST as the other results shared similar characteirstics. The figures show how the prediction results were changed - patched or broken. Although NEURECOVER shows modest numbers of patched inputs, it keeps numbers of broken inputs low. Arachne tends to repair the target label a lot but instead has radical regressions in another label (label 5 for CIFAR-10 and label 0 for Fashion-MNIST).

For retraining, regressions, or broken inputs, appear in various labels. It is notable that retraining has *shuffling effect*: many improvements and regressions occur at the same time even for the same label, e.g., label 4 for Fashion-MNIST. This behavior is very critical when we consider intensive assurance activities to check risks of failed inputs even if the total accuracy remains similar or better.

Answer to RQ3

NEURECOVER outperforms the baseline of Arachne by stably suppressing regressions also in repairing specific failure types. NEURECOVER outperforms retraining in suppressing regressions. Retraining tends to have more diverse regressions often with large shuffling of success and failure inputs.

D. Discussion

All the experimental results suggest the key benefits of NEURECOVER lie in its controllability in repair outcome, specifically, the capability to suppress regressions. On the other hand, repair performance in terms of RR is modest compared with Arachne or retraining. We can say NEURECOVER is conservative not to make destructive changes that introduce large regressions even if they introduce more improvements.

We argue this feature is significant when the impact of failures is large in safety-critical or quality-sensitive applications. In such cases, engineers and stakeholders are more careful to check whether each of failed inputs is acceptable in terms of safety, ethics, or other risks that affect trust on the target system. Regressions, even with a larger number of improvements, require costly recheck on the newly introduced failures. In this case, the conservative approach of NEURECOVER easily leads to modest but acceptable updates.

ΔACC values were sometimes around zero or negative for all the methods, especially in experiments for RQ3 with tight repair requirements. This fact suggests that the repair tasks intrinsically involve trade-offs and we cannot have a silver ballet to have improvements without any regressions. We thus assume that achieving high ACC values for whole the dataset alone is not the goal and it is necessary to argue impacts of specific success or failure types.

We had an informal workshop with industry practitioners from more than ten companies to discuss significance of considering fine-grained repair tasks such as the label-wise one for RQ3. All the practitioners agreed with the significance and showed concrete examples of repair requirements as follows.

- Specific failure types with worst performance should be fixed (the experimental setting of RQ3).
- Some labels are more significant than others, e.g., misrecognizing “stop” signs to something else is very critical.
- Some failure types are more critical than others, e.g., misrecognizing something to ”go ahead” signs is very critical.

The conservative approach of NEURECOVER has potentials to deal with such fine-grained requirements as partially shown in experiments for RQ3.

Applicability

NEURECOVER is suitable when failures are critical and engineers have costly tasks to check regressions for safety or trust assurance and/or when there are fine-grained requirements to prioritize labels or failure types.

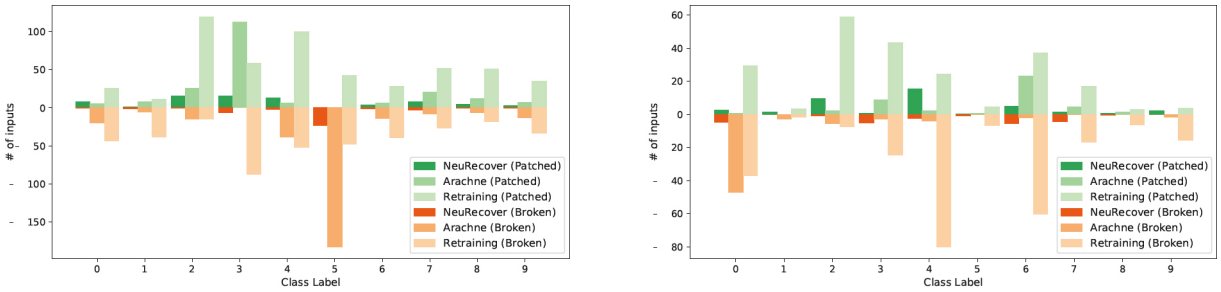
E. Threats to Validity

The core threat to the internal validity is that we did not include experiments over different optimization methods. In addition to PSO, we can consider using many other optimization methods, such as genetic algorithms and gradient descent.

The core threat to the external validity is the quantity of evaluation objects in the experiments. The experiment uses three datasets and three CNN architectures for image classification. It will be necessary to increase the number and types of datasets, models, and ML tasks to show that our technique is not dependent on a specific experimental object.

V. RELATED WORK

Many works on DNN testing and debugging have been inspired by software engineering techniques. One of these techniques is automated program repair (APR), which generates patches that make buggy programs pass all test cases. Some APR techniques achieve high repair performance by



(a) CIFAR10/VGG16

(b) Fashion-MNIST/VGG16

Fig. 3. RQ3. Label-wise repair performance

using the code editing history as a hint for repair [15], [24]. Spectrum-based fault localization, which is a part of APR, provides a score for suspiciousness by regarding program elements executed more frequently in failed test cases as more suspicious. In the same way for fault localization, the code editing history can be used to improve the accuracy of localizing faults [25], [26]. Our approach is motivated by the success of many history-based debugging methods.

Retraining is the most popular approach to fixing DNNs. Studies of test data generation techniques for DNNs have shown that retraining with adversarial examples generated as test data can improve robustness [27]. Several techniques have been proposed to fix specified failures in general, not just adversarial examples. *Few-Shot Guided Mix* (FSGMix) [28] is an augmentation-based repair technique that augments retraining data with the guidance of limited failure data. Srivastava et al. proposed a model learning scheme that adds a compatibility penalty to the loss function [29]. Yan et al. also proposed a retraining method that suppresses negative flip by adding penalty term based on model distillation to the loss function [30]. MODE is a debugging method for DNNs that works by identifying the features that are most affected by failed tests and generating inputs that are focused on those features by using a generative adversarial network (GAN).

We referred to Arachne [17] as the baseline method with the same approach of directly manipulating DNN weight parameters. Apricot [31] is another technique to obtain hints from different versions of DNNs created by using subsets of the training data. This approach rather captures how to fix the behavior for negative input data.

Continual Learning and similar techniques can be referred to as an effort to maintain deep learning model performance [32], [33]. They aim at reducing catastrophic forgetting (interference) [34], [35] enabling a deep learning model to learn a new task keeping previously learned tasks [36]. De Lange et al. classified Continual Learning into three categories [32]: *Replay methods*, *Regularization-based methods* and *Parameter-isolation methods*. Among them, several works are in common with NEURECOVER in an aspect of modifying specific parameters with intention to maintain performance.

Parameter-isolation method include techniques that incorporate isolating parameters such as branching multiple versions of a DNN model corresponding to each task or fixing specific weights during training [37]. Mallya and Lazebnik proposed PackNet to fix important parameters for one task, updating only the rest of the parameters. The task-oriented updates are repeated on sequential tasks [38].

Each *Parameter-isolation method* above is similar to NEURECOVER because NEURECOVER extracts parameters concerned to classification faults (fault localization steps); however, NEURECOVER prevents a deep learning model from degrading performance with a single task. On the other hand, works related to Continual Learning attempt to maintain the model performance in regard of different tasks.

VI. CONCLUSION

In this paper, we have presented a novel DNN repair method NEURECOVER by using the training history. The proposed method outperforms the existing repair method with the same approach of search-based repair owing to the capability to stably suppress regressions. We also demonstrated our method is especially effective when the repair requirements are tight by requesting to fix specific failure types and to avoid regressions. The presented approach is suitable for safety-critical or quality-sensitive applications that require intensive assurance activities including risk evaluation of failure cases as well as consideration of fine-grained requirements to prioritize labels or failure types. We believe this work demonstrated the significant first-step for fine-grained, regression-aware, and controllable engineering of DNNs.

The following issues are future works for NEURECOVER.

- Implementation of a technique for identifying suspicious weights of convolutional layers
- Study on repair techniques for other DNN models besides the image classification problem

ACKNOWLEDGMENT

This work was partly supported by JST-Mirai Program Grant Number JPMJMI20B8, Japan.

REFERENCES

[1] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[2] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1724–1734.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: towards real-time object detection with region proposal networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 1*, 2015, pp. 91–99.

[4] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 conference on empirical methods in natural language processing*, 2015, pp. 1422–1432.

[5] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[6] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[7] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2722–2730.

[8] K. D. Julian, J. Lopez, J. S. Brush, M. P. Owen, and M. J. Kochenderfer, "Policy compression for aircraft collision avoidance systems," in *2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC)*. IEEE, 2016, pp. 1–10.

[9] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning: A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 291–300.

[10] F. Ishikawa and N. Yoshioka, "How do engineers perceive difficulties in engineering of machine-learning systems?-questionnaire survey," in *2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*. IEEE, 2019, pp. 2–9.

[11] K. Hamada, F. Ishikawa, S. Masuda, M. Matsuya, and Y. Ujita, "Guidelines for quality assurance of machine learning-based artificial intelligence," in *SEKE2020: the 32nd International Conference on Software Engineering & Knowledge Engineering*, 2020, pp. 335–341.

[12] A. Karpathy, "Software 2.0," 2017. [Online]. Available: https://karpathy.medium.com/software-2-0-a64152b37c35

[13] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," *Advances in neural information processing systems*, vol. 28, pp. 2503–2511, 2015.

[14] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[15] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 648–659.

[16] K. Noda, Y. Nemoto, K. Hotta, H. Tanida, and S. Kikuchi, "Experience report: How effective is automated program repair for industrial software?" in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 612–616.

[17] J. Sohn, S. Kang, and S. Yoo, "Search based repair of deep neural networks," *arXiv preprint arXiv:1912.12463*, 2019. [Online]. Available: http://arxiv.org/abs/1912.12463

[18] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proc. of ICNN'95*, vol. 4. IEEE, 1995, pp. 1942–1948.

[19] A. Windisch, S. Wappler, and J. Wegener, "Applying particle swarm optimization to software testing," in *Proc. of GECCO'07*. Association for Computing Machinery, 2007, pp. 1121–1128. [Online]. Available: https://doi.org/10.1145/1276958.1277178

[20] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[21] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition," *Neural networks*, vol. 32, pp. 323–332, 2012.

[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.

[23] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[24] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[25] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 273–283.

[26] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[27] X. Gao, R. K. Saha, M. R. Prasad, and A. Roychoudhury, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *Proc. of ICSE'20*. IEEE, 2020, pp. 1147–1158.

[28] 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 *Proc. of ICSME'20*. IEEE, 2020, pp. 717–721.

[29] M. Srivastava, B. Nushi, E. Kamar, S. Shah, and E. Horvitz, "An empirical analysis of backward compatibility in machine learning systems," in *Proc. of KDD'20*, 2020, pp. 3272–3280.

[30] S. Yan, Y. Xiong, K. Kundu, S. Yang, S. Deng, M. Wang, W. Xia, and S. Soatto, "Positive-congruent training: Towards regression-free model updates," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 14 299–14 308.

[31] H. Zhang and W. Chan, "Apricot: A weight-adaptation approach to fixing deep learning models," in *Proc. of ASE'19*. IEEE, 2019, pp. 376–387.

[32] D. Hassabis, D. Kumaran, C. Summerfield, and M. Botvinick, "Neuroscience-inspired artificial intelligence," *Neuron*, vol. 95, no. 2, pp. 245–258, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0896627317305093

[33] Z. Chen and B. Liu, "Lifelong machine learning, second edition," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 12, no. 3, pp. 1–207, 2018. [Online]. Available: https://doi.org/10.2200/S00832ED1V01Y201802AIM037

[34] R. M. French, "Catastrophic forgetting in connectionist networks," *Trends in Cognitive Sciences*, vol. 3, no. 4, pp. 128–135, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1364661399012942

[35] M. McCloskey and N. J. Cohen, "Catastrophic interference in connectionist networks: The sequential learning problem," ser. Psychology of Learning and Motivation, G. H. Bower, Ed. Academic Press, 1989, vol. 24, pp. 109–165. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0079742108605368

[36] S. Thrun and T. M. Mitchell, "Lifelong robot learning," *Robotics and Autonomous Systems*, vol. 15, no. 1, pp. 25–46, 1995, the Biology and Technology of Intelligent Autonomous Agents. [Online]. Available: https://www.sciencedirect.com/science/article/pii/092188909500004Y

[37] M. Delange, R. Aljundi, M. Masana, S. Parisot, X. Jia, A. Leonardis, G. Slabaugh, and T. Tuytelaars, "A continual learning survey: Defying forgetting in classification tasks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021.

[38] A. Mallya and S. Lazebnik, "Packnet: Adding multiple tasks to a single network by iterative pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.