An Automated Monitoring and Repairing System for DNN Training

Xiaoyu Zhang, *Student Member, IEEE,* Chao Shen, *Senior Member, IEEE,* Shiqing Ma, Juan Zhai, and Chenhao Lin, *Member, IEEE,*

Abstract—With the widespread adoption of machine learning models, especially deep neural networks (DNNs), as an integral part of new intelligent software, the new tools to effectively support the model engineering and debugging process have received extensive attention. However, the existing tools only provide limited support for the training process. They are either post-training tools that fail to detect problems timely, resulting in wasting time and resources on training buggy models, or merely collecting the training data and still require manual analysis. In this paper, we propose AUTOTRAINER, an automated monitoring and repairing system for DNN training, which provides real-time monitoring for the model training process and automatically repairs eight commonly seen training problems. AUTOTRAINER monitors the training process and detects potential training problems. For any detected problem, AUTOTRAINER tries to fix it with the built-in state-of-the-art solutions. Our experiments on six datasets and 701 models show that the problem detection accuracy of AUTOTRAINER reaches 100% without false positives. Moreover, it fixes 98.42% of all detected problems and improves the model accuracy by 36.42% on average.

Index Terms—Machine learning security, deep learning debugging, deep learning repairing, deep learning training

1 INTRODUCTION

MACHINE Learning (ML) techniques that power the intelligent components of a software system are playing an increasingly significant role. The development of the Deep Neural Network (DNN) brings software intelligence broader prospects with its recent advances. It is estimated that the global edge AI software market is predicted to grow from $1460 million in 2021 to $8050 million in 2027 [2]. The COVID-19 pandemic further accelerates the application and deployment of Deep Learning (DL) techniques in various fields. Google and Harvard Global Health Institute have released improved COVID-19 Public Forecasts based on AI techniques to provide a projection of COVID-19 cases [3]. Microsoft Azure Health Bot service helps hospitals classify patients and answer questions about symptoms [4].

With this growing trend, DL techniques are studied in a wide range of industries, and the DL components powered by DNN models have become an integral part of the software. Unfortunately, like other software programs, DNN models also have bugs and vulnerabilities, which can severely affect the model performance and eventually lead to a waste of resources and security risks in the software system. Even worse, most domain developers and experts have limited or no knowledge of DL, leaving them unable to effectively solve the problems in the intelligent components of the software. This raises a great challenge for the community in debugging and repairing the DNN model and its development process.

To address this challenge, the researchers have developed some tools that can help the developers who are new to the DL techniques [5–7]. For instance, MODE [5], proposed as a novel model debugging method, identifies the buggy neurons that bring about unexpected model behaviors and repairs the model performance by selecting additional training examples. These works focus on detecting and repairing the models whose training has been completed and are known as the *post-training* techniques. However, these existing techniques can hardly automatically test and fix the model problems, and their deployment and usage often require expert experience in the ML techniques, which limits their value in real-world scenarios. Furthermore, we observe in our experiments that model training problems occur randomly during the training process (as shown in §3), and the post-training tools cannot detect and fix model problems in a timely manner during model training. As a result, developers have to check the buggy model after the training is completed, which leads to a waste of time and computing resources.

Moreover, the existing DNN frameworks have provided limited support for detecting and monitoring the potential model problems in the training procedure. TensorBoard [8], which is the default debugging toolkit of the TensorFlow [1] framework, provides records and visualization for various values during model training, such as model weights and gradients. There are some other similar tools, such as Visdom [2], Manifold [3] and PyTorch Profiler [4]. These toolkits can track and visualize metrics like model gradients and

- *This paper is an extended version of work [1]. Chao Shen is the corresponding author.*
- *Xiaoyu Zhang, Chao Shen and Chenhao Lin are with the School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an 710049, China.*
  E-mail: zxy0927@stu.xjtu.edu.cn, {chaoshen, linchenhao}@xjtu.edu.cn.
- *Shiqing Ma and Juan Zhai are with the Manning College of Information & Computer Sciences, University of Massachusetts, Amherst, United States.*
  E-mail: {shiqingma, juanzhai}@umass.edu.

1. https://github.com/tensorflow/tensorflow
2. https://github.com/facebookresearch/visdom
3. https://github.com/uber/manifold
4. https://pytorch.org/docs/stable/profiler

visualize the change in the collected data during the training with diagrams. Similar to traditional software debuggers (e.g., Microsoft Visual Studio Debugger), these DNN debugging tools can help developers understand the model performance and training status but cannot analyze the collected data and fix model problems.

The nature of the training problem makes it difficult for these debuggers to discover these problems in time. Our previous work [1] reveals that the occurrence of a training problem is highly random. Firstly, whether a training problem occurs or not is random even for the same training script. Secondly, the moment when a training problem occurs during the model training is also random. We provide real cases to illustrate these findings in §3. Considering the fact that many DNN training tasks may take days or even months, it is impractical for developers to constantly monitor and analyze the model training data and manually diagnose the potential problems. Additionally, these runtime tools cannot provide automated analysis for the massive amounts of training data, let alone apply the solutions to repair the problems. To address the aforementioned limitations, there is an urgent need for a tool that frees developers from manually monitoring the model training process and automatically diagnoses and fixes problems during training, thereby increasing developer productivity and efficiency and improving the reliability and security of intelligent software systems.

Although our earlier work [1] provides an automatic training problem monitoring and repairing approach, it lacks a sufficient and comprehensive evaluation to assess the specific repair effects of the solutions as well as support for several other common problems, which can also severely impact the model training performance and lead to security risks in the DNN models. This paper presents an extended version of our previous work [1]. Our previous work mainly focuses on the training problems related to the backward propagation of gradients in model training (e.g., vanishing gradient problem). This paper extends and explores three new training problems in forward propagation, such as the improper output activation function problem. To effectively repair these training problems, we propose and implement extra repair methods and supplement the design and implementation details in this paper. Moreover, we design a new research question and supplement more experimental models to comprehensively evaluate the effectiveness of our system. In a nutshell, this paper presents AUTOTRAINER, an automated monitoring and repairing system, which currently focuses on eight common training problems(i.e., vanishing gradient, exploding gradient, dying ReLU, oscillating loss, slow convergence, improper output activation function, improper loss function, and abnormal data). Taking a model and its training configuration (e.g., hyperparameter, optimizers) as the input, AUTOTRAINER starts the training and monitors and records training data like loss value and gradient. During the monitoring, AUTOTRAINER conducts regular analysis to diagnose potential training problems. When the symptoms of any training problem are detected, AUTOTRAINER will try to repair it with the built-in solutions and restart the model training procedure. During the retraining, if another problem is detected, AUTOTRAINER will regard the old problem as resolved and attempt to repair

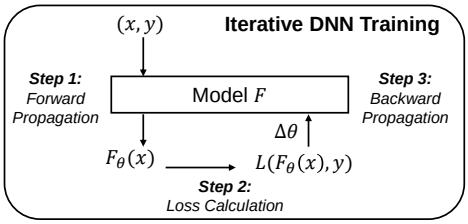


Fig. 1: The Basic Steps of the Iterative Model Training Process

the new problem. If there is no problem arises during retraining, AUTOTRAINER will consider that all problems are resolved, and then it delivers a well-trained model with the corresponding configuration to the user. If AUTOTRAINER fails to solve this problem, it will notify the user with a completed training log. Our contributions are:

- We summarize and formalize definitions for the symptoms of eight common training problems.
- We propose a novel approach to monitor in real-time and automatically repair eight different training problems during model training.
- We develop a prototype system AUTOTRAINER based on the proposed idea and evaluate it with six public datasets and 701 models. The evaluation results demonstrate that AUTOTRAINER can effectively detect all 506 problems for 422 buggy models and repair 498 problems of them with a ratio of 98.42%. On average, the test accuracy can be improved from 41.79% to 78.21% (1.87 times the original accuracy).
- Our implementation, collected models, configurations, experiment results, and problem solutions are publicly available at [9].

The rest of this paper is organized as follows. We first introduce the background in §2 and then presents two motivation examples of training problems in §3. The design and implementation of AUTOTRAINER are explained in §4. We show and analyze the experiment results in §5. Subsequently, §6 discusses the limitations of AUTOTRAINER. Then, after presenting the related work in §7, we discuss the insight of this work in §8 and conclude in §9.

2 BACKGROUND

2.1 DNN Model Training

A DNN model is a parameterized function $F_\theta : X \mapsto Y$, where $x \in X$ is an $m-$dimensional input (i.e., $x \in \mathbb{R}^m$) and $y \in Y$ is the corresponding output label. θ represents the weights that determine the outputs of the model F. A DNN with n layers can be formally represented as a composite function $F = l_n \circ l_{n-1} \circ \cdots \circ l_1$, where l represent a layer in the model. The input layer l_1 takes raw inputs and passes them on to the subsequent layers (i.e., the hidden layers). The following hidden layers extract the features of the input, and the output layer l_n is trained to predict the output based on the extracted features. The output of each layer l can be expressed as $F_l = \sigma(\theta_l * F_{l-1} + b_l)$, where b_l is the bias values of layer l. σ, which is known as the activation function, defines the specific output of layer l with a given input(§2.3). The connection between two consecutive layers l and $l-1$ in the model can be represented by a set of

matrices, which is referred to as the weight θ_l. The weight matrices of each layer in the model need to be initialized before training. After initialization, with a large set of input-output pairs (x_i, y_i) given, the training process updates all weight parameters θ of the DNN model to minimize the differences between a predicted result $F_\theta(x)$ and the corresponding ground truth label y. The loss function $\mathcal{L}(F_\theta(x), y)$ is used to measure this difference. Therefore, the essence of the DNN model training is to minimize the value of $\mathcal{L}$.

Specifically, training a DNN model consists of three steps, as shown in Fig. 1. Firstly, the *forward propagation* step uses the existing weight θ to predict output labels $F_\theta(x)$ for the training samples x. Then, the *loss calculation* computes the loss function value $\mathcal{L}$ based on predicted output and ground truth labels. Subsequently, the *backward propagation* step adjusts and updates the weight values $\Delta\theta$ from the output layer l_n back to the input layer l_1, trying to minimize the difference using an optimization method. The optimization method is usually a gradient descent algorithm or its variants, such as stochastic gradient descent (SGD). The training process will repeat the above steps until the stopping criteria are reached, which means the difference converges to a minimum value or the maximum number of training iterations is reached.

2.2 Gradient Descent

DNN model training leverages loss functions to quantify the prediction ability of a DNN model and uses the gradient descent algorithm and its variants to guide the update of model weights. The algorithm tweaks the weights in the opposite direction to the gradient of the loss function and adjusts the weights to appropriate values. Specifically, each weight has an update proportional to the partial derivative of the loss function for the current weight. The gradients are typically computed by automatic differentiation techniques utilizing the chain rule. Therefore, the computation of gradients for weight has the effect of multiplying many numbers that are from subsequent layers.

In general, the predictive ability of a DNN model is closely related to the number of layers (also known as the model depth). Increasing the number of layers can enable a neural network to train on a large-scale training dataset and effectively learn more complex mapping functions from inputs to outputs. However, blindly adding the model layers can lead to abnormal gradients that negatively impact the model training performance. Two common related training problems are *vanishing gradient* and *exploding gradient*.

Problem 1 (Vanishing Gradient Problem)**.** *In backward propagation, when the gradient is computed by multiplying many small numbers, the gradient can be vanishingly small, especially for layers close to the input layer. Consequently, the model weights will hardly change and the loss function may end up with a very large value, meaning the model will have a low accuracy after training. Such a problem is referred to as vanishing gradient (VG).*

Symptoms of VG. During the backward propagation from the output layer to the input layer, the gradient decreases exponentially and approaches zero in layers close to the input layer. The model accuracy remains low during training.

Problem 2 (Exploding Gradient Problem)**.** *In contrast to VG, the gradient can grow exponentially as it propagates backward. This can lead to unexpectedly large values or even NaN values in gradients, resulting in poor model accuracy. Such a problem is referred to as exploding gradient (EG).*

Symptoms of EG. The gradient increases exponentially from the output layer to the input layer during backward propagation. It can grow large or even overflow to *NaN* value in the layers close to the input layer. Similar to the vanishing gradient problem, the model accuracy is also low during training.

2.3 Activation Function

For a given set of inputs, each neuron in DNN computes the weighted sum and then adds a bias to the sum. Then, an activation function processes the computed sum and produces an output for the neuron. The activation function determines how much the input is relevant for the following stage, guiding the network to leverage important features and suppress irrelevant features. Common activation functions include Sigmoid, ReLU (Rectified Linear Unit), etc. Taking the ReLU activation function as an example, it is one of the most widely-used nonlinear activation functions in the neural networks [10–12]. For input x, the output of the ReLU activation function can be represented as $ReLU(x) = max\{x, 0\}$. Existing work [13] has demonstrated its excellent training effect, which can improve model sparsity and achieve better convergence in training. However, ReLU has its own limitations, among which *dying ReLU* is the most common and serious one.

The activation function significantly affects the ability and performance of a DNN model, and different activation functions can be used in different parts of the model. In general, the hidden layers and output layers in the models use different activation functions. The former is designed to process the features or data from the previous layer and pass the output to the following layer, while the latter performs the prediction task of the model based on the output of the hidden layer. Therefore, the activation function of the hidden layer often uses nonlinear activation functions (e.g., ReLU) to introduce the non-linearity into the model and get access to the rich hypothesis space [14]. The output activation function and the loss function, on the other hand, depend on model tasks [15]. With the advancement of DL security, researchers have designed customized loss functions and the like to accomplish some domain-specific tasks, including improving model robustness and fairness [16, 17].

We introduce three related training problems here, namely *dying ReLU* and *improper output activation function* and *improper loss function*, respectively.

Problem 3 (Dying ReLU Problem)**.** *When a ReLU neuron receives a non-positive input, it will output zero, making the neuron inactive. In such cases, the neuron is very likely to remain inactive forever since a gradient-based optimization algorithm will not tweak the weights of an inactive neuron. Consequently, these neurons cannot be leveraged to distinguish between the prediction and ground truth, and if there are many such neurons, it may end up with a large part of the neurons in the network contributing nothing to the prediction task. This is known as dying ReLU (DR).*

Symptoms of Dying ReLU. When training a DNN with ReLU as the activation function of the hidden layers, the gradients of a large percentage of the neurons are zero and the model accuracy is low during training.

Problem 4 (Improper Output Activation Function Problem)**.** *The activation function of the output layer doesn't match the prediction task of the DNN model. As a result, the DNN model cannot perform the prediction task correctly based on the extracted features from the hidden layers, and the training may end up with a stalled high loss value and low accuracy. Such a problem is referred to as improper output activation function (IO).*

Symptoms of IO. The activation function of the output layer in the DNN model does not match the prediction task, and the model accuracy remains low during training.

Problem 5 (Improper Loss Function Problem)**.** *Similar to the IO problem, the loss function doesn't match the prediction task of the DNN model, resulting in low accuracy during training. We refer to such a problem as improper loss function (IL).*

Symptoms of IL. The loss function of the DNN model does not match its prediction task, and the model accuracy is low during training.

2.4 Convergence

The training goal is to reduce the loss value to a minimum. To determine the point of convergence, there are usually two conditions. One is that the training time has reached the maximum allowed iteration (defined by the user). The other one is that the training accuracy has reached the desired values. Some training cases may end up with a set of low accuracy models even after the maximal number of training iterations. The model often struggles to learn valuable features from the training data in these cases, resulting in the training not converging properly. They are usually caused by three problems: *oscillating loss*, *slow convergence*, and *abnormal data*.

Problem 6 (Oscillating Loss Problem)**.** *It is inevitable for the loss value to go up and down during the training procedure. However, large changes in the loss value occur without the decreasing trend, and the training may not converge for a very long time, which should be enough for training the model. Such a problem is referred to as oscillating loss (OL).*

Symptoms of OL. The training accuracy keeps fluctuating in a large range for a long time.

Problem 7 (Slow Convergence Problem)**.** *The loss value is high and decreases so slowly that no significant accuracy improvement has been made, and the model training may end up with low accuracy when the maximal number of training iterations is finished. We refer to such a problem as slow convergence (SC).*

Symptoms of SC. The training accuracy holds a low value for a long time and the loss is decreasing slowly.

Problem 8 (Abnormal Data Problem)**.** *The training data is not preprocessed or normalized, resulting in too large values or even NaN values, which will affect the output prediction and weight updating of the model. It is difficult for the model to process the input or learn features from the training data. We refer to such a problem as abnormal data (AD).*

Symptoms of AD. The output values of the layers near the input layer can become large or even *NaN* value during the forward propagation, and the model accuracy remains low.

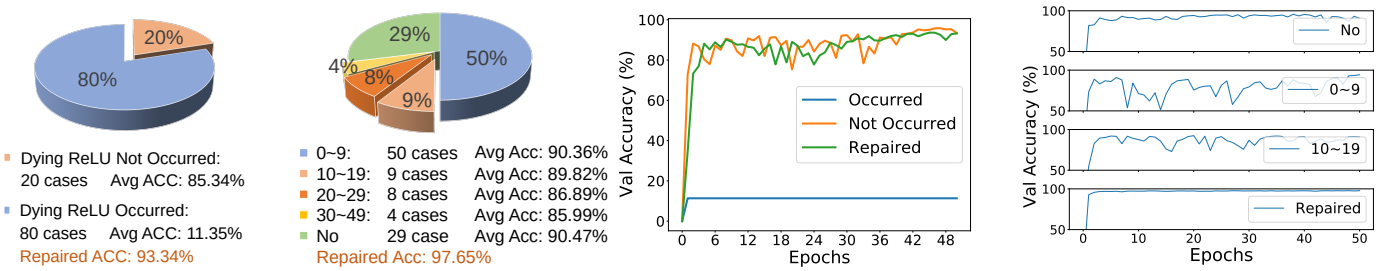
3 Identifying DNN Problems During Training

To our knowledge, there is no existing tool that provides real-time support for users to identify the aforementioned DNN problems during training. The TensorBoard Debugger [8] in TensorFlow and the PyTorch Hooks [18] in PyTorch can help users inspect the changes of program variables (e.g., loss value, gradient) after training. However, analyzing these recorded data and fixing the potential training problems in the models requires expert knowledge. Even though many of these problems are common in DNN training, and their symptoms and solutions have been studied and analyzed, identifying the problem and finding an appropriate solution is far from a piece of cake. In this paper, we propose AutoTrainer, a DNN training tool that can automatically monitor DNN internal values (i.e., neuron outputs and gradients), loss values, and training accuracy values during the training procedure and inspect possible problems. If a problem is identified, AutoTrainer will try to fix it automatically. AutoTrainer is designed as an automated real-time training monitoring and fixing tool due to the random nature of the training problems. Specifically, even for the same training script, whether a training problem occurs or not is random during training. When a training problem occurs, which iteration it occurs and affects is also random. In the following sections, we provide two cases to illustrate the randomness of the training problem and motivate our design.

3.1 Training problem occurrence is highly random

When we train a model with the same configuration and training data multiple times, it is random whether the training problem occurs during a single training procedure. The reason for this phenomenon lies in the random values used in the model training process, such as random weight initialization and stochastic gradient descent. Influenced by random values, during repeated training, a training problem can arise in several cases and severely affect model accuracy. But in other cases, the training problem doesn't occur, and the model achieves high accuracy after training.

Here we provide a case with *dying ReLU* problem in training to illustrate the randomness of the training problem discussed above. We build a DNN model with 34 layers (650,000 parameters) and use ReLU as the activation function of the hidden layers. For the training configuration, we choose MNIST [19] handwritten digit dataset (50,000 training and 10,000 testing samples) as the training dataset, Adam [20] as the optimizer, and set the learning rate to 0.001, which is recommended by TensorFlow. Following the prior work [21], we set the maximal number of the training epochs to 50 (i.e., over 80,000 training steps) to show the impact of dead neurons on model training. The detailed configuration and model are shown in our repository [9]. Subsequently, We train the model with this training configuration 100 times. Fig. 2(a) shows the distribution of the occurrence



(a) Occurrence **(b)** Time of Occurrence **(c)** Occurred VS Not Occurred **(d)** Cases of Occurrence Time

Fig. 2: Problems Occurrence and Time of Occurrence are Random

of the *dying ReLU* (DR) problem in the repeated training. In 80% of these training trials, we can observe the DR problem that severely affects the training performance and leads to the average accuracy of 11.35%. For the remaining 20% of training attempts, the model accuracy increases quickly, and the weights update continuously during training. They achieve an average accuracy of 85.34% when the training is completed. Fig. 2(c) provides a intuitive comparison. Blue, orange, and green separately represent the accuracy curves of training that the DR problem occurred, didn't occur, and is repaired by AUTOTRAINER. Note that both of the curves are from the same training script and model. We can observe that when the DR problem occurs, the training stops converging in the first few epochs, and the update of the model weights stalls, resulting in an accuracy of only 11.35%. In contrast, the training without DR problem finally achieves over 95% accuracy. The difference between the two training results indicates that whether a problem occurs or not is random. In addition, we also use AUTOTRAINER to repair the DR problem in this case, and the model accuracy is improved to 93.23% after repair.

3.2 The time when a training problem occurs is random

Similar to the randomness of problem occurrence, the time when a problem actually occurs is also random during training. We illustrate this phenomenon with an example of a DNN model that suffers from the *oscillating loss* (OL) problem. This model contains 20 layers and also uses ReLU as the activation function of the hidden layers. The corresponding training configuration also uses the Adam optimizer with a learning rate of 0.001 in training Following the prior work [21, 22], we set the maximal number of the training epoch to 50 to observe the OL problem. More Details are in our repository [9]. The distribution of the stages (epoch number) when the OL problem occurs is shown in Fig. 2(b). We can observe that, for 29% of the cases, the OL problem is not triggered, and they reach an average accuracy of 90.47%. But in half of the cases, the problem occurs in the first ten epochs, and the percentages of detecting the problem in other stages are separately 9%, 8%, and 4%. This result demonstrates at which iteration or stage of the training a particular problem occurs is random. Fig. 2(d) shows a comparison of the training where OL problems occurred at different stages. Four cases are shown from top to bottom: OL did not occur, occurred on 0-9 and on 10-19 epochs, and is repaired by AUTOTRAINER. In the first case where OL does not occur, the accuracy curve is relatively stable and finally achieved 90.65%. Comparing the second and third cases, we can see that the earlier the training problem occurs, the greater the fluctuation of the subsequent training curve, and the more severe the training is affected.

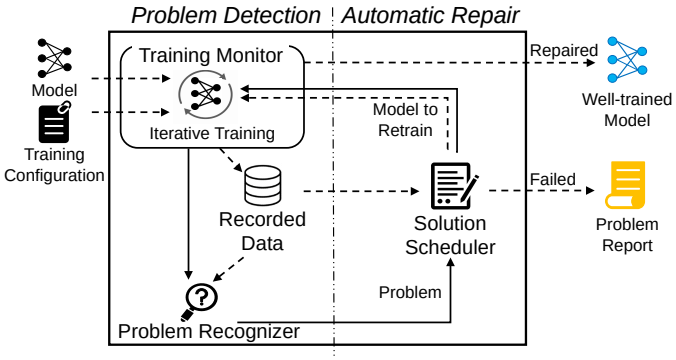


Fig. 3: Overarching Design of AUTOTRAINER

For training problems that cannot be predicted whether and when they occur, the real-time training monitor in AUTOTRAINER can acutely detect training problems. For this model, AUTOTRAINER can detect the problem at the early stage (i.e., before 20 epochs out of 50) in the wide majority of cases (i.e., more than 80% of the cases where the problem occurs). In contrast, existing *post-training* methods do not collect real-time data, making them unable to provide timely detection during the training. After detection, our system attempts to resolve the problem with four built-in solutions (See §4.4). After the repair of AUTOTRAINER, the OL problem in this model has been successfully alleviated, and the training finally achieves an accuracy of 97.65%, as shown in the fourth case of Fig. 2(d).

4 SYSTEM DESIGN

Fig. 3 gives the overarching design of our system, which consists of two cooperating components, namely the *problem detection* module (left) and the *automatic repair* module (right). The solid line with arrows in this figure represents the control flow in AUTOTRAINER, and the dashed line with arrows represents the data flow. The whole system starts by training a model with an initial training configuration and using the problem detection module to monitor the training iterations. During this process, the monitor records the training data, and the problem recognizer analyzes the recorded data and detects the potential training problems. When a problem is detected, the system launches the automatic repair module and attempts to fix the problem with built-in solutions. Subsequently, AUTOTRAINER retrains the repaired model with new training settings and continues to

monitor the new training until it can be completed without any problems occurring (repaired) or the detected problems that cannot be resolved (failed).

AUTOTRAINER takes the training configurations (e.g., loss function, optimizer, and learning rate), the original model, and user preferences as inputs. The user preferences are configurable parameters for AUTOTRAINER, which include the thresholds to judge the problems, preferred repair solutions, and so on. AUTOTRAINER has a set of default values for them, and the user can adjust them according to their needs. Details will be presented in §4.2. The problem detection module monitors the training information like loss value, gradients, and layer outputs. During this, the problem recognizer is triggered periodically, analyzing the recorded data to recognize symptoms and determining whether a training problem exists in the model training. If a problem is detected, AUTOTRAINER will use the automatic repair module to address it. Otherwise, the training monitor will output the trained model with its training configurations to the user, as shown in the blue output in Fig. 3.

For each problem, AUTOTRAINER has a few built-in solutions to fix them. However, one solution may not work in all cases. If there is a detected problem that is the same as before, it means the applied solution cannot solve the problem for this particular case. Hence, the solution scheduler will retrieve the next one, apply it, and restart training. If a new problem is detected from the repaired training, the solution scheduler will save the current model into a checkpoint for subsequent attempts to select the corresponding solutions to continue solving the new model. AUTOTRAINER has a default order for the problem solutions, designed to prioritize the optimal performing fixes to solve the training problems as quickly as possible. The order of solutions can also be reorganized by users. If none of the solutions can fix the problem, the solution scheduler will report a failed case with the whole training log to the users, as the yellow part shown in Fig. 3. It is worth noting that for failed repairs of the new problem detected from a repaired training, AUTOTRAINER will select the model with the highest accuracy as a part of outputs based on the repair history and checkpoints.

4.1 Training Monitor

The training monitor starts a training procedure and records data which is used to recognize symptoms and retrain the model when a problem is detected. The recorded data can be divided into two categories, namely the static data and the runtime data.

The static data is the data that does not change during the whole model training process. It is generally determined and stored at the beginning of model training. The static data includes:

- Model definition including layers and their configurations (e.g., kernel sizes in convolutional layers).
- Optimization method definition and its parameters.
- Hyper-parameters and other necessary variables used in training, such as the batch size and learning rate.

The runtime data refers to the data that the model will continuously generate and update during training, such as the gradients and the training accuracy. The runtime data includes:

- Training accuracy and loss values.
- Calculated gradients for each neuron.
- Outputs and weights for each model layer.

Note that the data of each training procedure will be recorded separately and can be queried by the user.

4.2 Problem Recognizer

The problem recognizer regularly analyzes the recorded data to recognize training problems. The symptoms leveraged to detect problems are formalized and shown in Table 1. The first column lists the training problems, and the second column specifies the symptoms involving gradient and training accuracy, etc. If the depicted condition is met, AUTOTRAINER regards the corresponding symptom as observed. When all symptoms of a problem are observed during training, AUTOTRAINER will determine that the problem exists and then terminate training. The last column presents the built-in solutions in AUTOTRAINER.

• **VG.** The symptoms of the VG problem can be formalized as two conditions. Firstly, there has not been a trained model whose accuracy is good enough to terminate the training ($max(Acc) \leq \Theta$). This check is by default enabled and checked by all existing DNN training platforms already. If there is such a model, the training should be terminated. Secondly, in the recent α_1 training iterations, the gradient has been dropped from layer to layer in the backward propagation, and the gradient becomes very small (smaller than a threshold value β_2). To measure the change and value of gradients, we use the l_2-norm, which is borrowed from existing literature in the AI research community [23–25].

• **EG.** The definition of the symptoms of EG is very similar to that of VG, except that the gradient is increasing from layer to layer in backward propagation, or it has already become NaN values in some layers (meaning that it cannot propagate back to the input layer already).

• **DR.** Dying ReLU means that there has been a set of neurons whose gradients have been 0 in the recent few iterations ($[k - \alpha_3, k]$), and this set is large and makes up a large portion of the entire DNN (more than a threshold value γ) while model accuracy is still low.

• **OL.** Intuitively, the symptom of an OL problem is that there have been a lot of oscillating loss values from the start till now. To measure if there are oscillating loss values, we first extract two lists of loss values, A and B, which represent the maximum optimal and minimal optimum loss values (in time order), respectively. Then, we calculate the degree of oscillation by computing the differences of a consecutive pair of elements in A and B. If the difference is larger than η, we think this is a significant oscillation, and when such oscillation occurs very frequently, we think there is an OL problem in the model training.

• **SC.** By definition, SC means the accuracy of trained models is growing slowly. To detect this problem, AUTOTRAINER checks the training accuracy change for the past iterations. If the change remains small during training, it indicates that the training has been trapped into a locally optimal point, and the training process has failed to improve it. Based on this, AUTOTRAINER determines that the SC problem occurs.

TABLE 1: Problem Symptoms and Repair Solution Candidates

Training Problem	Symptom	Solution
Vanishing Gradient [23–25]	Gradient: $\forall i \in [k-\alpha_1, k], \frac{\left\|G^i_{l_2}\right\|}{\left\|G^i_{l_3}\right\|} \dots \frac{\left\|G^i_{l_{n-1}}\right\|}{\left\|G^i_{l_n}\right\|} \leq \beta_1 \bigwedge \left\|G^i_{l_2}\right\| \leq \beta_2$ Accuracy: $max(Acc) \leq \Theta$	S1: Adding Batch Normalization Layers S2: Substituting Activation Functions
Exploding Gradient [23–25]	Gradient: $\forall i \in [k-\alpha_2, k], \frac{\left\|G^i_{l_2}\right\|}{\left\|G^i_{l_3}\right\|} \dots \frac{\left\|G^i_{l_{n-1}}\right\|}{\left\|G^i_{l_n}\right\|} \geq \beta_3 \bigvee \exists j \in N, G^i_j = NaN$ Accuracy: $max(Acc) \leq \Theta$	S1: Adding Batch Normalization Layers S2: Substituting Activation Functions S3: Adding Gradient Clip
Dying ReLU [21]	Gradient: $\forall i \in [k-\alpha_3, k], \frac{\lvert\{j \in N \rvert, G^i_j = 0\}\rvert}{\lvert N \rvert} \geq \gamma$ Accuracy: $max(Acc) \leq \Theta$	S1: Adding Batch Normalization Layers S2: Substituting Activation Functions S4: Substituting Initializer
Oscillating Loss [22]	Accuracy: $\frac{\lvert\{i \in [1, min(\lvert A \rvert, \lvert B \rvert))] \rvert A[i] - B[i] \geq \zeta\}\rvert}{k} \geq \eta$	S4: Substituting Initializer S5: Adjusting Batch Sizes S6: Adjusting Learning Rate S7: Substituting Optimizer
Slow Convergence [26]	Accuracy: $\forall i \in [1, k], \lvert Acc[i] - Acc[i-1] \rvert \leq \delta$	S4: Substituting Initializer S6: Adjusting Learning Rate S7: Substituting Optimizer
Improper Output Activation Function [15]	Activation Function: $\sigma_n \notin \text{ActSet}(T((x, y), O^k))$ Accuracy: $max(Acc) \leq \Theta$	S2: Substituting Activation Functions
Improper Loss Function [15]	Loss Function: $L \notin \text{LossSet}(T((x, y), O^k))$ Accuracy: $max(Acc) \leq \Theta$	S8: Substituting Loss Function
Abnormal Data [27]	Layer Output: $\forall i \in [k-\alpha_4, k], \exists j \in n, \left\|O^i_{l_j}\right\| \geq \omega \bigvee \exists m \in N, O^i_m = NaN$ Accuracy: $max(Acc) \leq \Theta$	S9: Processing Input Data

1 G^b_a, the gradient of layer a in iteration b
2 n, the number of layers of a DNN
3 N, all neurons of a DNN
4 k, the current training iteration
5 $\alpha_1/\alpha_2/\alpha_3$, thresholds for iterations
6 $\beta_1/\beta_2/\beta_3$, thresholds for gradients
7 Θ, the training accuracy threshold
8 γ, the threshold for the percentage of neurons with 0 gradients
9 δ, the threshold for accuracy difference
10 ζ, the threshold for the difference of maximum and minimum optimal
11 η, the threshold for the percentage of times of large loss fluctuation
12 Acc, accuracy arry for each iteration
13 max, the maximum function
14 A/B, arrays of maximum/minimum optimal
15 O^b_a, the output of layer a in the last batch of iteration b
16 $ActSet/LossSet$, recommended activation and loss function sets
17 T, the model task from analysis

• **IO.** The improper output activation function problem means that the model uses the inappropriate activation function in the output layer to make the prediction which leads to a low accuracy during training. Therefore, to determine this problem, two symptoms need to be present. Firstly, the activation function σ_n of the output layer doesn't match the model prediction task T, which can be represented as $\sigma_n \notin \text{ActSet}(T)$, where $ActSet(T)$ indicates the output activation functions recommended by existing work [15] for the given task. For example, a model uses the ReLU activation function in the last layer to make predictions for multi-classification tasks but not the Softmax activation function recommended by existing work. The second condition is that the model accuracy is not high enough to terminate the training (i.e., $max(Acc) \leq \Theta$).

• **IL.** Similar to the IO problem, there are also two symptoms of improper loss function problem. Firstly, the loss function L does not match the prediction task of the model T, which can be represented as$L \notin \text{LossSet}(T)$, where $LossSet(T)$ indicates the recommended loss functions [15]. Secondly, the model accuracy remains low in training.

• **AD.** In some cases, the training data is not properly preprocessed or normalized [27]. For example, the values of the CIFAR-10 dataset are generally normalized to $[-1, 1]$ and not $[0, 255]$. Anomalous values (e.g., NaN and Inf) also need to be cleaned and processed before being inputted into the model. When the model uses abnormal input data during training, the training process may be difficult to converge, and the loss function cannot be reduced. Therefore, the symptoms of the AD problem have two parts. First of all, there are abnormal values in layer outputs of the model, which is manifested in that the maximum output value of a certain layer exceeds the given threshold ω or it has already become NaN values in some layers. Here we use the l_∞-norm to measure the maximum value of layer outputs. Secondly, the model accuracy remains low in training.

Note that although it seems like it is possible to detect and fix the forward propagation problems (i.e., IL, IO, and AD) before training, this strategy is unreliable. On the one hand, for customized loss functions, data, etc., it is difficult to design a method to determine whether there are anomalous before training starts. With the application of DL, researchers proposed a large number of domain-specific tasks that require customized data, loss function, and activation function, for example, improving model fairness [16, 28] and robustness [17], and knowledge distillation [29]. It is difficult to analyze the customized implementations in advance and determine whether they are suitable for the target training tasks. On the other hand, deep learning models are complex systems with millions of parameters, and it is difficult to intuitively predict whether the training results converge or not before training. To accurately identify these training problems, the problem recognizer first analyzes and determines the current model's task (e.g., multi-classification) based on the model's training data and intermediate outputs of forward propagation, and then evaluates to identify the training problem based on the model loss, activation function, and accuracy, etc.

4.3 Solution Scheduler

The main role of the solution scheduler is to pick one solution to fix the problem and restart the training procedure. For the same problem, it will try each possible

solution one by one based on the default order if users do not specify preferred orders. If one solution can fix the problem, the scheduler will not be triggered by the same problem. Otherwise, it will try a new solution. If none of these solutions can fix it, AUTOTRAINER fails to resolve this problem and will report this to the user to determine what to do next. Note that the model training may suffer from multiple problems. These training problems have different symptoms and exposure stages that do not occur at the same time. AUTOTRAINER attempts to solve the detected problems one by one in the order of exposure.

All solutions in AUTOTRAINER have been summarized from existing work and evaluated in large-scale experiments, and they are ordered based on the experimental results. This ordering is designed to prioritize the most effective repair strategy to resolve the training problem as quickly as possible. The experiments in §5.5 verify the effectiveness of the built-in repair methods in AUTOTRAINER. Moreover, AUTOTRAINER provides the interface for users to customize their repair strategies and sequence.

4.4 Existing Solutions

The existing research has proposed some methods to alleviate or solve the training problem. Unfortunately, there is no silver bullet and one solution cannot be guaranteed to work for all cases. As shown in the third column in the Table 1, for each problem, AUTOTRAINER collects a few possible solutions that have proved the effectiveness in the prior study and our evaluation. AUTOTRAINER has implemented a total of 9 repair methods, as shown in the following:

• **S1: Adding Batch Normalization Layers.** Batch normalization is a method that normalizes the neuron values of a layer by re-centering and re-scaling them. This helps eliminate the unexpected gradient and neuron activation values. Specifically, the normalization will squeeze the values into a specific range, and as such, the gradient updates will not vanish or explode during the backward propagation and reduce the possibility of getting inactive neurons [30, 31]. We follow the prior work [30] and implement our solution to add batch normalization before activation function layers.

• **S2: Substituting Activation Functions.** As aforementioned, ReLU is a commonly adopted activation function. The gradient of ReLU activation is 1 when the input is greater than 0, meaning the gradient will remain the same without decreasing or increasing dramatically with the proper optimizer and learning rate. Hence, substituting the current activation function of the hidden layers with ReLU and its variants (e.g., SELU [32], LeakyReLU [33]) can mitigate both the vanishing gradient problem and the exploding gradient problem. For the IO problem, AUTOTRAINER will substitute the activation function of the output layer based on the suggestions of existing work [15]. For example, for a binary classification task and a model using the ReLU activation function in the output layer, AUTOTRAINER will select the Sigmoid activation to replace it after detecting the improper output activation problem.

• **S3: Adding Gradient Clipping.** Gradient clipping clips gradient values outside of a specified range, which essentially restricts the update of the weight value to a limited region. By removing obviously large gradient values, it can alleviate the exploding gradient in the models [15, 34, 35]. Following the prior work [34, 35], AUTOTRAINER clips the gradient of each layer to $[-10, 10]$.

• **S4: Substituting Initializers.** Initializers give initial values to the model weights and set a starting point for the optimization process. Thus, inappropriate initialization can severely affect the mode training performance and may lead to several training problems. Lu et al. [36] propose that the popular initialization schemes like He Initialization [37] suffer from the Dying ReLU problem. Xavier initialization [38] is proposed to solve the oscillating loss and slow convergence problem by initiating the weight values to a proper range. Thus, AUTOTRAINER also tries to substitute the used initializers when a model encounters the Dying ReLU, slow convergence, or oscillating loss problems.

• **S5: Adjusting Batch Sizes.** Batch size is the number of training samples used in one iteration to estimate the layer outputs and gradients. A large batch size may cause the loss value to fall into a poor local minimum, while a small batch size might lead to oscillating loss [39, 40]. Existing work [34, 35, 40] has analyzed and evaluated the impact of different choices of batch size for DNN training. Following the prior work [40, 40, 41]. AUTOTRAINER first initializes the batch size to 32. When the oscillating loss problem occurs, AUTOTRAINER doubles the batch size until it reaches 256.

• **S6: Adjusting Learning Rates.** The learning rate determines the amount of change to the model in each update (i.e., each backward propagation). If the learning rate is too large, the weights are likely to have a fluctuating update and the loss value will oscillate and even increase over training epochs [15]. In this case, decreasing the learning rate can be helpful to tackle the oscillating loss problem during training. Generally, a small learning rate allows the model to learn more optimal or even globally optimal weights with the risk of taking a very long time to finish the training. At one extreme, the training may never converge to a low loss value even after the maximal number of training epochs. Therefore, one candidate solution for the slow convergence problem is increasing the learning rate [15]. In AUTOTRAINER, the values for learning rates depend on the different optimizers they use. We follow the suggestions made by their original authors (e.g., Adam [20]) and existing empirical evidence [41, 42]. Specifically, we choose 0.01 for SGD-based optimizers, and 0.001 for Adam and other adaptive optimizers. If the slow convergence problem still exists, AUTOTRAINER increases it 10 times; and if the oscillating loss problem still exists, AUTOTRAINER decreases it by a factor of 10.

• **S7: Substituting Optimizers.** Optimizers are algorithms used to update weights to reduce the loss value. An optimizer can behave differently in different scenarios. Practically, a substitution of an optimizer can help address various training problems. Stochastic Gradient Descent (SGD) [43] computes an estimated gradient on a randomly selected small subset of data samples instead of computing an actual gradient on the entire dataset. Based on the rationale, the weights are updated more frequently in SGD which can speed up the convergence but may cause fluctuations in the loss value. Momentum [44] is a method introduced to speed up the SGD optimizer and alleviate loss oscillations. It works by adding a fraction of the update in

the past timesteps to the current update. Following prior work, we set the momentum value to 0.9, which means the weights will update based on 90% of the previous gradient and 10% of the new gradient. Adaptive Moment Estimation (Adam) [20] is another widely adopted optimizer that uses momentum and adaptive learning rates that gradually decrease the learning rate in training. It can prevent the training from missing the local minimum and accelerate the convergence. In a nutshell, AUTOTRAINER uses optimizers with momentum to alleviate the oscillating loss problem and randomly selects a different optimizer with a default learning rate to alleviate the slow convergence problem.

• **S8: Substituting Loss Function.** During training, the value of the loss function reflects the error between the predicted results and the ground truth results, guiding the optimizer to update the model weights in the direction of reducing the error. Choosing an appropriate loss function can efficiently guide the training process and speed up the convergence. The existing work points out that the loss function value is generally related to the model output unit and the model task [15]. For example, the prediction tasks of binary classification generally use the Sigmoid activation function in the output layer and cooperate with the binary cross-entropy loss function. When the improper loss function problem occurs, AUTOTRAINER will select a new loss function according to the model task.

• **S9: Processing Input Data.** The abnormal input values can severely affect the training process and may even cause the unexpected termination of training [27]. In addition, the too large values in the raw input data may cause abnormal changes in gradients during model training, resulting in decreasing the convergence speed of training. For the training suffering from the abnormal data problem, AUTOTRAINER will normalize the input data and remove outliers to improve the training performance. In addition, the interface of AUTOTRAINER makes it easy to update data processing methods according to the state-of-the-art work in the field of data science [45, 46].

5 EVALUATION

The prototype of AUTOTRAINER is implemented on top of TensorFlow 2.3.0 [47] and also supports the models built in the TensorFlow 2.1.0 version. In the evaluation, we aim to answer the following research questions:

RQ1: Can AUTOTRAINER effectively detect and repair training problems?

RQ2: Is the overhead of AUTOTRAINER acceptable in detecting and repairing training problems?

RQ3: How do different configurable parameters in AUTOTRAINER affect the performance?

RQ4: Can the built-in solutions in AUTOTRAINER effectively repair training problems?

5.1 Setup

We performed our experiments on six popular datasets: Circle [48], Blob [49], MNIST [19], CIFAR-10 [50], IMDB [51] and Reuters [52]. Circle and Blob are two datasets from SKLearn [53] for classification tasks, which contain two and three categories, respectively. MNIST is a gray-scale image dataset used for handwritten digit recognition. CIFAR-10 is a widely used colored image dataset used for object recognition and contains 10 categories. IMDB is a movie review dataset for sentiment analysis with two classes. Reuters is a newswire dataset for document classification.

In total, we collected 701 models and their training configurations with various DNN model structures (CNN, RNN and fully connected (FC) layers only) for these six datasets. The model information is shown as follows.

- Only FC layer models are trained on Blob and Circle datasets. Their number of layers ranges from 7 to 19, with a minimum of 6 FC layers and the number of trainable parameters ranges from 141 to 513.
- CNN models are trained and evaluated on CIFAR-10 and MNIST models. They contain 2 or more convolutional layers and 3 or more FC layers and the largest one has 28 layers and 544,810 trainable parameters. There is also a number of the CNN model that uses the LeNet-5 architecture [54] to illustrate the negative impact of the training problem.
- RNN models consist of 2 or more LSTM or RNN layers and 3 or more FC layers. The largest one contains 21 layers and 5,692,354 parameters and the smallest one has 3 layers and 3,362,478 parameters.

Among them, 422 of them have suffered from at least one training problem and the rest are benign models. Some of the models are collected and reproduced from reported buggy models on GitHub, StackOverflow, existing papers, and personal blogs [39, 55–57]. The other models are gathered from machine learning experts within our organization. Moreover, we have manually verified whether these models exhibit training problems. Specifically, two co-authors with expertise in SE and AI security are invited to verify the collected models. They record the training logs to verify whether the models have the same abnormal behavior and training problems described by the authors (e.g., NaN loss [55]). For those models that lack problem descriptions, we use the symptoms and thresholds described in §4.2 for evaluation and verification. According to our statistics, verifying a single model takes about 10 minutes, and each participant takes about four weeks to complete all verification. Subsequently, following the prior work [58, 59], we use Cohen's Kappa statistic to measure the level of agreement (inter-rater reliability) of the annotation results of two participants, which is 0.92 (i.e., "strong agreement" [60]). For inconsistency, we invite a third co-author to moderate the discussion and conduct verification until we obtain results that are recognized by all three participants. To ensure the reliability of the experiment results, all models and their training configurations and all experiment results are available in our repository [9].

If not specified, all experiments in this section are conducted on a server with Intel(R) Xeon E5-2620 2.1GHz 8-core processors, 130 GB of RAM and an NVIDIA TITAN V GPU running Ubuntu 20.04 as the operating system.

5.2 Effectiveness of AUTOTRAINER

Experiment Design: To evaluate the effectiveness of AUTOTRAINER, we train the 701 collected models with

TABLE 2: The Overall Problem Repaired Results

Dataset	VG		EG			DR			SC			OL		IO	IL	AD	Total
	S2	*S1*	*S2*	*S1*	*S3*	*S2*	*S1*	*S4*	*S7*	*S6*	*S4*	*S7*	*S6*	*S2*	*S8*	*S9*	
Blob	10	2	10	0	0	4	3	1	31	0	0	4	0	28	13	20	126
Circle	9	1	9	1	0	6	3	0	48	1	0	7	1	9	9	8	112
CIFAR-10	7	0	7	1	0	2	1	0	27	1	0	2	0	13	7	10	78
MNIST	6	2	10	0	0	4	0	0	20	1	0	7	1	7	13	9	80
Reuters	0	4	6	0	0	-	-	-	19	7	0	0	4	6	15	-	61
IMDB	5	3	5	0	1	-	-	-	9	3	0	0	4	3	8	-	41
Total	37	12	47	2	1	16	7	1	154	13	0	20	10	66	65	47	498
Repaired		49			50			24			167		30	66	65	47	498
Failed		4			4			0			0		0	0	0	0	8
Total		53			54			24			167		30	66	65	47	506

their training configurations with AUTOTRAINER to observe whether AUTOTRAINER can effectively detect and solve the training problems. Due to the randomness in performing these experiments, we run the training five times to ensure that the problems have been exposed, and the average problem discovery rate is 95.50%. To evaluate the effectiveness of AUTOTRAINER, we start two parallel training processes for the same model. The two training processes share the usage of the same random number including the initialization weights. They also share the same set of training hyper-parameters and optimization methods. During training, we collect training logs including gradients, loss values, etc.

Results: For all 422 buggy models, we detect a total of 506 training problems as some models have more than one. We display partial results in Table 3. The first two columns separately list the datasets and the model status with the corresponding number of models. The "Repaired" indicates that a model has been successfully repaired by AUTOTRAINER and there is no training problem in it now, and "Failed" indicates that the problem still exists even after AUTOTRAINER has tried all built-in solutions. We use the "Normal" to denote models without training problems. For ease of exposition and analysis below, the third column of Table 3 gives the serial number (from 1 to 701) of all models in the experiment, and the fourth column shows the number of detected problems of each model. The following columns represent the accuracy, training time, and memory consumption of the corresponding model. To avoid the potential bias and shortcomings of a single accuracy metric on multi-classification tasks, we also record the precision and recall in experiments. Due to space limitations, we only include the recall and precision improvement for models after repair in Table 3. The full table and detailed results are in our repository [9]. The columns "Orig." and "AT" separately show the results for the original model training and repaired model training. "Ratio" gives the ratio between the values of a repaired model and the corresponding original model, and the column "Improve" shows the absolute improvement in accuracy, recall, and precision that our system achieves. The cells in purple separately correspond to the models with the highest accuracy improvement, maximum training overhead, and maximum memory overhead, while the cells in grey separately correspond to the ones with the least accuracy improvement, minimum training overhead, and minimum memory overhead. In addition, the red cells show the averaged results of the repaired training on six datasets.

To evaluate the repair effects, we also calculated the number of problems that are fixed by individual solutions and the accuracy improvement, as shown in Table 2, Fig. 4, and Table 4. The columns in Table 2 present the problem

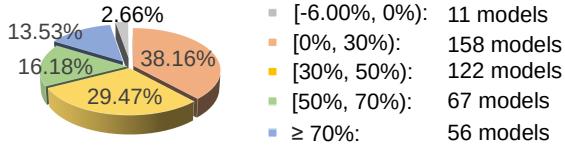


Fig. 4: Accuracy Change After Model Training Repair.

and corresponding solutions whose orders are their default priority used for repairing in AUTOTRAINER. Each number in the top half of the table denotes the number of problems that are repaired successfully by the corresponding solution. The bottom half summarizes the number of problems of different statuses. Moreover, the pie chart in Fig. 4 demonstrates the distribution of change ranges in accuracy with corresponding numbers of models. The columns "#Repaired" and "Avg. Improvement" in Table 4 separately show the number of repaired training problems and averaged accuracy improvement on different datasets. The columns "Backward" and "Forward" indicate those training problems that occur during *backward propagation*, and those that occur during *forward propagation* and *loss calculation*, respectively.

Analysis. The experiment results demonstrate the effectiveness of AUTOTRAINER. Firstly, AUTOTRAINER can effectively detect the defined training problems with a 100% success rate on all the 701 model training, and none of the benign training is misclassified as problematic. Secondly, AUTOTRAINER can effectively repair the buggy training procedures. After successful repair, it can improve the accuracy by 36.42%, and its maximum accuracy improvement in the experiment is 90.17%. In addition, the successful repair on the six datasets resulted in increments of 36.27% and 44.52% on recall and precision. The repair results shown in these two metrics are similar to the repair results in terms of accuracy, further demonstrating the effectiveness of AUTOTRAINER in fixing problems and improving model performance across the board without introducing potential bias. Notice that when EG occurs, it may result in NaN values in the model weights, leading to NaN output results for all inputs. In such cases, we do not measure the prediction accuracy and directly report them as "n.a." in Table 3.

From Table 2, we observe that AUTOTRAINER is able to respectively repair 92.45% and 92.59% of VG and EG problems, and it fixes all the DR, SC, OL, IO, IL, and AD problems in buggy training trials. Table 2 and Table 4 demonstrate that AUTOTRAINER is capable of handling different types of datasets, model problems, and model architectures and achieves significant repair effects. Regarding SC and OL, we find that the first two solutions (i.e., S7 and S6) can effectively solve all the problems we encountered in the evaluation, so the other solutions are not used in the experiments. Therefore, we conduct an evaluation in §5.5 to evaluate the actual effect of each repair solution.

Fig. 4 shows the distribution of accuracy improvement after repairing. Specifically, there are 56 repaired models that improve the accuracy by at least 70%, and over 50% of the models improve accuracy by more than 30%. Combined with the average accuracy of repaired models, we can observe that a significant number of the models have a very low original accuracy of around 10%. We manually analyze

TABLE 3: Overall Results of AUTOTRAINER

Dataset	Status	No.	#Prob.	Accuracy				Recall	Precision	Train Time			Average Memory		
				Orig.(%)	*AT(%)*	*Imp.(%)*	*Ratio*	Imp.(%)	Imp.(%)	*Orig.(s)*	*AT(s)*	*Ratio*	*Orig.(MB)*	*AT(MB)*	*Ratio*
Blob	Repaired: 92	1	1	10.33	83.00	72.67	8.03	81.33	83.85	6.19	11.45	1.85	1239.07	1239.25	1.00
		2	2	75.00	69.00	-6.00	0.92	73.33	27.28	13.34	125.11	9.38	1320.49	1319.74	1.00
		3	3	30.67	83.33	52.67	2.72	5.81	50.02	15.86	586.63	36.98	1564.95	1565.12	1.00
		4	1	40.33	78.67	38.33	1.95	25.00	35.83	17.99	24.03	1.34	1550.91	1554.17	1.00
		5	1	59.00	80.00	21.00	1.36	54.67	-7.21	13.41	23.75	1.77	1157.88	1174.61	1.01
		6	1	50.33	82.67	32.33	1.64	-56.67	56.94	12.99	30.99	2.39	1282.06	1258.63	0.98
		Avg.	1.25	52.68	79.51	26.83	1.51	24.83	36.97	13.49	39.47	2.92	1377.56	1378.2	1.00
	Failed: 2	93	1	33.67	33.67	0.00	1	0.00	0.00	16.03	109.56	6.84	1576.94	1574.19	1.00
		94	1	33.67	33.67	0.00	1	0.00	0.00	6.89	86.64	12.57	1574.7	1573.95	1.00
		Avg.	1	33.67	33.67	0.00	1	0.00	0.00	11.46	98.1	8.56	1575.82	1574.07	1.00
	Normal: 53	95	-	50.67	-	-	-	-	-	15.61	17.74	1.14	1505.91	1486.04	0.99
		96	-	45.67	-	-	-	-	-	15.71	16.94	1.08	1514.26	1509.71	1.00
		Avg.	-	77.18	-	-	-	-	-	16.41	18.08	1.1	1477.09	1476.88	1.00
Circle	Repaired: 98	148	1	0.00	86.33	86.33	n.a.	86.09	86.67	18	28.44	1.58	1320.03	1321.28	1.00
		149	1	87.33	82.67	-4.67	0.95	63.58	-20.24	22.98	71.85	3.13	1325.53	1326.22	1.00
		150	2	49.67	78.00	28.33	1.57	-5.94	43.62	3.5	57.18	16.35	1332.55	1332.32	1.00
		151	1	54.33	85.00	30.67	1.56	25.17	40.76	60.88	74.27	1.22	1358.89	1358.89	1.00
		152	1	56.67	67.67	11.00	1.19	19.87	6.59	10.43	20.55	1.97	1161.57	1177.07	1.01
		153	1	50.00	85.67	35.67	1.71	31.13	35.72	22.96	41.97	1.83	1281.22	1258.49	0.98
		Avg.	1.11	50.24	81.14	30.90	1.61	25.02	30.91	24.89	57.27	2.3	1313.57	1313.61	1.00
	Normal: 39	246	-	86.67	-	-	-	-	-	16.2	16.94	1.05	1305.43	1302.58	1.00
		247	-	68.00	-	-	-	-	-	29.14	30.53	1.05	1313.31	1306.58	0.99
		Avg.	-	79.60	-	-	-	0.00	0.00	25.27	30.68	1.21	1269	1334.09	1.05
CIFAR -10	Repaired: 73	285	1	8.25	70.29	62.04	8.52	51.10	73.68	56.69	85.3	1.5	3778.36	3661.16	0.97
		286	1	70.56	68.91	-1.65	0.98	-10.44	-0.74	218.11	345.07	1.58	7641.48	9068.34	1.19
		287	2	10.00	71.73	61.73	7.17	67.56	76.64	73.48	1180.53	16.07	4477.29	3774.16	0.84
		288	1	10.02	66.12	56.10	6.6	38.76	67.48	108.6	98.34	0.91	7576.53	9463.46	1.25
		289	1	10.00	43.96	33.96	4.40	20.12	66.60	187.96	248.83	1.32	4694.23	6400.61	1.36
		290	1	10.00	67.43	57.43	6.74	54.95	77.93	214.44	295.44	1.38	3589.40	2817.81	0.79
		Avg.	1.08	18.52	63.33	44.80	3.42	42.86	61.85	201.6	389.2	1.93	4972.65	4912.53	0.99
	Failed: 2	358	1	10.00	10.00	0.00	1	0.00	0.00	382.79	492.6	1.29	3777.13	3777.36	1.00
		359	2	10.00	10.00	0.00	1	0.00	0.00	186.52	419.96	2.25	6316.59	6301.32	1.00
	Normal: 35	360	-	65.37	-	-	-	-	-	342.88	319.32	0.93	4452.31	3749.12	0.84
		361	-	56.56	-	-	-	-	-	236.45	244.89	1.04	3749.78	3753.79	1.00
		Avg.	-	63.48	-	-	-	-	-	145.63	146.39	1.01	3946.77	3826.03	0.97
MNIST	Repaired: 65	395	1	8.89	99.06	90.17	11.14	98.75	98.84	28.69	36.99	1.29	3523.83	4080.47	1.16
		396	1	99.16	99.29	0.13	1	-0.08	-0.11	119.67	166.53	1.39	3588.25	3174.85	0.88
		397	2	9.80	98.99	89.19	10.1	98.86	99.11	365.55	2294.69	6.28	3283.42	3283.67	1.00
		398	1	9.80	98.73	88.93	10.07	98.46	98.88	99.99	101.78	1.02	3265.32	2916.77	0.89
		399	1	98.39	99.18	0.79	1.01	0.11	0.04	34.67	48.84	1.41	2930.73	3814.31	1.30
		400	1	97.56	98.82	1.26	1.01	-2.98	-1.59	40.31	76.37	1.89	3996.26	3342.43	0.84
		Avg.	1.08	43.55	98.82	55.26	2.27	63.71	60.10	179.19	358.84	2	3228.46	3204.64	0.99
	Normal: 80	460	-	98.79	-	-	-	-	-	173.85	173.47	1	3203.96	3198.19	1.00
		461	-	86.54	-	-	-	-	-	135.93	136.42	1	3168.69	2925.68	0.92
		Avg.	-	96.94	-	-	-	-	-	227.69	227.94	1	3258.77	3198.95	0.98
Reuters	Repaired: 51	540	2	0.53	66.38	65.85	124.25	62.33	74.23	308.96	946.55	3.06	4857.65	4879.49	1.00
		541	1	59.75	59.71	-0.04	1	-0.22	-0.65	928.16	1834.58	1.98	2534.15	2575.85	1.02
		542	1	47.91	50.22	2.32	1.05	9.75	-12.50	1278.41	4778.25	3.74	2312.99	2360.14	1.02
		543	1	0.53	62.91	62.38	117.75	56.54	80.03	1316.38	1321	1	1859.9	1869.22	1.01
		544	1	50.49	67.59	17.10	1.34	15.72	-8.54	220.45	350.06	1.59	3684.76	4059.10	1.10
		545	1	0.53	57.52	56.99	107.67	48.17	86.70	2591.11	4363.19	1.68	1823.66	1758.23	0.96
		Avg.	1.02	29.44	61.09	31.65	2.08	27.07	28.89	813.64	1682.75	2.07	3621.92	3655.3	1.01
	Failed: 1	591	1	36.02	36.02	0.00	1	8.68	-1.64	1484	1460.37	0.98	1881.61	1820.14	0.97
	Normal: 41	592	-	37.13	-	-	-	-	-	1466.05	1510.32	1.03	1912.03	1927.08	1.01
		593	-	36.15	-	-	-	-	-	1400.59	1471.97	1.05	1901.38	1930.53	1.02
		Avg.	-	51.08	-	-	-	-	-	1001.36	1022.2	1.02	3071.84	3103.06	1.01
IMDB	Repaired: 35	633	1	0.00	87.08	87.08	n.a.	87.41	87.41	3982.2	8876.97	2.23	1998.46	2069.23	1.04
		634	1	85.51	83.64	-1.87	0.98	-2.90	-2.90	1446.27	2060.99	1.43	2264.55	2348.56	1.04
		635	1	49.33	86.03	36.70	1.74	37.88	37.88	670.69	6925.99	10.33	2256.18	2342.93	1.04
		636	1	49.12	85.22	36.10	1.73	36.10	36.10	1069.43	1307.58	1.22	2258.64	2345.51	1.04
		637	1	85.81	86.82	1.01	1.01	-0.39	-0.39	641.24	990.92	1.55	4029.30	4514.27	1.12
		638	1	50.00	86.11	36.11	1.72	35.39	35.39	3703.41	9246.12	2.50	2057.98	1854.65	0.90
		Avg.	1.03	52.70	84.33	31.63	1.6	30.82	29.20	2270.36	5026.83	2.21	3027.69	3097.92	1.02
	Failed: 3	668	1	0.00	0.00	0.00	n.a.	49.96	49.96	130.46	2257.56	17.3	2261.88	2348.53	1.04
		669	1	50.00	50.00	0.00	1	0.00	0.00	1062.77	1480.36	1.39	2182.29	2349.58	1.08
		670	1	0.00	0.00	0.00	n.a.	50.00	50.00	2089.21	1451.98	0.69	2184.26	2044.23	0.94
		Avg.	1	16.67	16.67	0.00	1	33.32	33.32	1484.15	1729.97	1.17	2209.47	2247.45	1.02
	Normal: 31	671	-	87.38	-	-	-	-	-	1951.87	1984.37	1.02	2260.38	2347.84	1.04
		672	-	83.14	-	-	-	-	-	2022.07	2036.58	1.01	2191.92	2088.6	0.95
		Avg.	-	78.97	-	-	-	-	-	1167.99	1176.08	1.01	3859.67	3916.31	1.02
Normal		Avg.	-	77.83	-	-	-	-	-	367.14	372.34	1.01	2767.78	2755.42	1.00
Repaired		Avg.	1.11	41.79	78.21	36.42	1.87	36.27	44.52	364.74	779.56	2.14	2702.91	2698.77	1.00
Failed		Avg.	1.13	21.67	21.67	0.00	1	12.33	11.04	669.83	969.88	1.45	2719.42	2723.63	1.00

TABLE 4: The Accuracy Improvement of the Repaired Problems

Dataset	#Repaired								Avg. Improvement(%)							
	Backward					Forward			Backward					Forward		
	VG	*EG*	*DR*	*OL*	*SC*	IO	IL	AD	*VG*	*EG*	*DR*	*OL*	*SC*	IO	IL	AD
Blob	12	10	8	4	31	28	13	20	27.26	29.13	18.33	-3.00	41.44	25.21	2.12	18.60
Circle	10	10	9	8	49	9	9	8	29.12	80.79	26.86	9.27	32.84	13.69	8.47	28.11
CIFAR-10	7	8	3	2	28	13	7	10	54.48	57.64	59.38	10.97	50.58	29.48	41.58	33.22
MNIST	8	10	4	8	21	7	13	9	87.41	87.21	86.02	56.80	82.77	75.03	0.45	0.73
Reuters	4	6	-	4	26	6	15	-	18.59	49.82	-	19.75	36.89	42.07	16.55	-
IMDB	8	6	-	4	12	3	8	-	36.43	85.94	-	-0.96	31.33	13.67	16.21	-
Total/Average	49	50	24	30	167	66	65	47	41.39	65.92	45.01	15.14	44.87	28.10	11.79	21.23

these models and the training history before and after repair and find that these models are affected by severe training problems such as VG and EG, which caused them to stop converging at an early stage of training. This illustrates, on the one hand, the severe impact of training problems on models, where the classification ability of the affected model is difficult to improve even after dozens of epochs, and on the other hand, the superiority of AUTOTRAINER in fixing training problems and improving model accuracy. We also notice that there are 11 models (out of all 414 models) whose accuracy has a slight reduction after being repaired. Our analysis finds that it is because they have other uncovered problems. How to detect and repair these bugs will be one of our future directions.

To further illustrate the repair effect of AUTOTRAINER, we provide four repaired cases on different datasets in Fig. 5. The blue and orange curves separately show the accuracy of the original and repaired models. For DR and VG problems in backward propagation (Fig. 5(a) and Fig. 5(b)), the models repaired by AUTOTRAINER can achieve accuracy over 50% in the first several epochs, and continue to improve the accuracy in the subsequent training. For IO and AD problems in forward propagation (Fig. 5(c) and Fig. 5(d)), AUTOTRAINER can repair models and significantly improve accuracy by replacing the activation function and processing abnormal data.

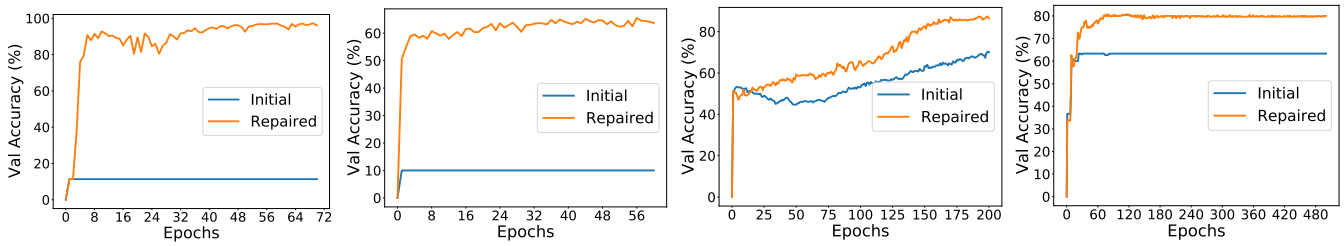
From Table 4 and Fig. 5, we can observe that the accuracy improvement of repair on those training problems related to forward propagation and loss calculation is usually smaller than the improvement in fixing training problems in backward propagation (e.g., EG). The average improvement of accuracy on repaired forward propagation problems is 20.33%, and the average accuracy boost on the repaired backward propagation problems is 44.85%. We manually analyze the repair process and results and find that the difference in repair effect mainly comes from the characteristics of different training problems. Training problems like VG, EG, and SC tend to lead to model training stagnation with very low accuracy (e.g., around 10% accuracy in Fig. 5(a)) due to gradients that are too small or too large to update weights. However, when IO, IL, and AD occur, the model can often still update the weights, but the model is unable to obtain loss function values to effectively guide the weight updates, and thus training is difficult to converge to an optimal, as shown in Fig. 5(d).

In repeated training trials, we observe that due to the random occurrence of training problems (§3.1), the same configuration may have different problem detection results. The reason is that some problems do not appear during training, and AUTOTRAINER can only detect remaining training problems or no problems. We collect the problem discovery rate of each model over five repeated trials and use the median (i.e., 5) to divide the training problems into two groups, namely *frequent problem* with a discovery rate higher or equal to the median and *occasional problem* with a discovery rate below the median. We then count the averaged accuracy improvement of the model after repairing frequent problems and occasional problems using AUTOTRAINER, as shown in the blue and green bars in Fig. 6. Moreover, for models with *occasional problem*, we record the accuracy improvement of training without being affected by problems compared to the problematic training (orange bar in Fig. 6). The frequent problems repaired by AUTOTRAINER achieve the highest accuracy improvement, reaching 36.14%. The repaired occasional problems show an accuracy increase of 8.54%. The significant difference in repair effect is due to the type of training problems. Frequent problems include all gradient-related problems (e.g., VG and EG), whose repair can result in an accuracy improvement of over 80%, as shown in Table 6. Occasional problems mainly consist of the OL problem which typically exhibits an accuracy increment of about 10% after repair. Additionally, the accuracy improvement from retraining models with occasional problems is only 2.71%, which is only 31.73% of the accuracy improvement from AUTOTRAINER. This indicates that retraining the model cannot effectively solve occasional problems, further demonstrating the effectiveness of the repair of AUTOTRAINER. In addition, the different detection results caused by occasional problems will eventually cause the final model to perform worse than the repaired model. Our repeated experiment ensures that all problems are detected and repaired by AUTOTRAINER.

Answer to RQ1: AUTOTRAINER is effective in detecting and fixing training problems. It can detect all training problems without false positives in the experiment and soon fix them one by one. In the experiment, AUTOTRAINER detects 422 buggy models with 506 problems from a total of 701 models. Then it repairs 414 models and corresponding 498 problems. Particularly, fixing the training problems in the backward propagation and forward propagation steps separately obtain the average accuracy improvement of 44.85% and 20.33%. The training problem repair rate reaches 98.42%, and the average model accuracy improves by 36.42% after repair.

5.3 Efficiency of AUTOTRAINER

Experiment Design and Results: To evaluate the efficiency of AUTOTRAINER, we train all 701 models with and without AUTOTRAINER enabled. During training, we separately collected the time used to train the model and the memory usage for both the original training and AUTOTRAINER. The experiments are conducted 5 times, and the overhead is calculated as the average of these 5 runs.



(a) DR Problem in Backward Propagation on MNIST Dataset **(b)** VG Problem in Backward Propagation on CIFAR-10 Dataset **(c)** IO Problem in Forward Propagation on Circle Dataset **(d)** AD Problem in Forward Propagation on Blob Dataset

Fig. 5: The Repair Results of AUTOTRAINER on Different Datasets

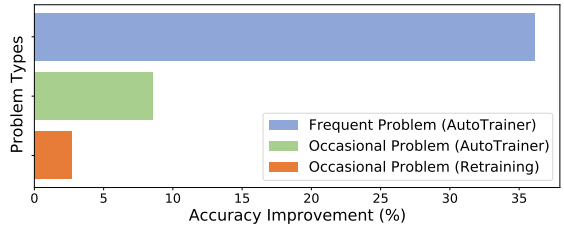
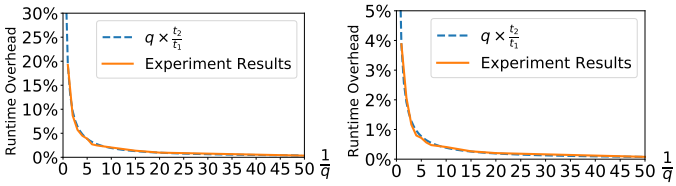


Fig. 6: Accuracy Improvement on Different Problems.



(a) Circle Dataset **(b)** MNIST Dataset

Fig. 7: Runtime Overhead vs. Problem Check Frequency.

Runtime Overhead Analysis: For normal training, the runtime overhead is purely from the problem checker, which is about 1%. From Table 3, we observe that the runtime overhead on smaller datasets is usually larger (e.g., Blob 8% vs. almost 0% for MNIST). This is because the total time for training on small datasets is relatively short, making the runtime overhead ratio larger. To handle buggy training in experiments, AUTOTRAINER takes 0.68 more training time on average. We perform a deeper analysis to understand the overhead of the individual components and find that, no matter whether training problems occur in the forward or backward propagation steps, retraining takes over 99% and the rest two parts (i.e., problem recognizer and repair) take less than 1%. It means that AUTOTRAINER only costs a little time ($\leq$ 1% total overhead) in automatically searching the suitable solutions for the problems, which requires lots of manual operations and is time-consuming in existing strategies. As discussed in §4.3, to repair a problem, it may try several times, which leads AUTOTRAINER to train several models. It is worth noting that, AUTOTRAINER implements a fixed problem recognizer and solution scheduler in problem detection and repair, therefore, there are no cases where the runtime overhead of detecting and selecting solutions for a particular training problem is significantly higher than that of other problems among the eight training problems.

Checking Frequency v.s. Runtime Overhead. More frequent problem checking causes higher runtime overhead. Suppose that one training iteration and one checking separately take t_1 and t_2 time, then the overhead of AUTOTRAINER is roughly $q \times t_2/t_1$, where q is the checking frequency. Fig. 7 presents the correlations between checking frequency and runtime overhead on Circle and MNIST datasets. The X-axis is the number of iterations between two checks ($1/q$), and Y-axis is the runtime overhead. The solid line represents the collected data and the dashed curve is the theoretical results (i.e., $q \times t_2/t_1$). As we can see, the shapes of experiment data conform to our theoretical analysis. By comparing the two figures, we observe the smaller dataset has a higher runtime overhead, which is consistent with our data in Table 3. By default, AUTOTRAINER checks the problem every 3 iterations, which causes less than 5% overhead even for small datasets like Circle, and for large datasets, the overhead will be less.

Lower frequency checking can reduce the overhead, but may cause a longer delay in detection. Table 5 shows the effects of different check frequencies on the training problem. Each row represents one problem type, and each column denotes a different check frequency. Numbers in cells show the delayed iterations between the occurrence of the problem and the detection of the problem. Considering VG, if AUTOTRAINER performs the checking every 15 iterations, it needs 6 extra iterations to detect it compared with checking problems every other iteration. For AD problems that occur in forward propagation, smaller detection frequencies also result in larger detection delays (from 0.35 to 6.56), ultimately leading to wasting time on buggy training.

Memory Overhead Analysis. AUTOTRAINER has very limited memory overhead (-1% to 1% on average in the experiments) since AUTOTRAINER reuses data that has been collected. To detect problems, AUTOTRAINER reuses the current gradient values, historical loss values, and training accuracy which are automatically collected and stored in training by all major frameworks. The overhead of AUTOTRAINER is mostly due to program variables, which are negligible compared with neuron and gradient values.

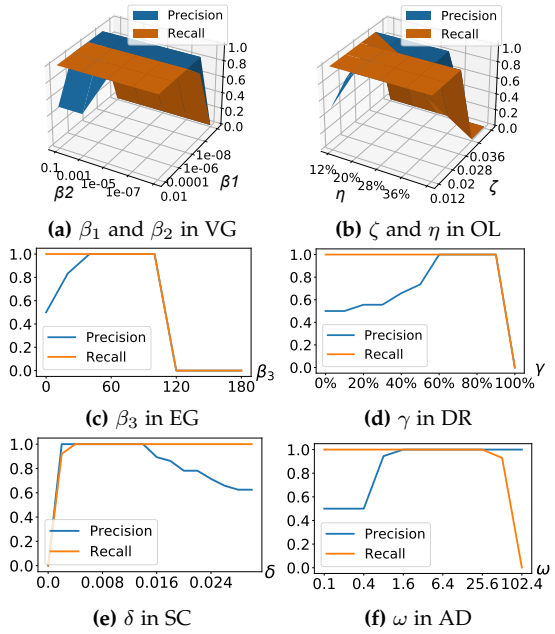
Answer to RQ2: AUTOTRAINER can detect and repair training problems efficiently. It brings an average memory overhead of about 1% of the original memory consumption. Moreover, 99% of the time overhead of AUTOTRAINER comes from model retraining, and the remaining 1% is used to detect problems and apply repair solutions, whether the training problem occurs in the forward or backward propagation.

5.4 Effects of Configurable Parameters

AUTOTRAINER leverages a set of configurable parameters to recognize the symptoms and detect problems. We conduct an evaluation to observe how each parameter affects the

TABLE 5: Check Frequency vs. Delay in Problem Detection

Problem	Detection Delay					
	1/q=2	1/q=3	1/q=4	1/q=5	1/q=9	1/q=15
VG	0.33	1.12	1.48	1.78	3.22	6.22
EG	0.38	1.38	2.38	3.38	7.38	13.38
DY	0.43	1.15	2.15	2.29	8.01	8.01
OL	0.40	1.60	1.60	2.40	3.40	6.40
SC	0.32	1.06	1.25	2.13	2.74	6.09
AD	0.35	1.17	1.62	2.21	3.01	6.56



(a) β_1 and β_2 in VG **(b)** ζ and η in OL

(c) β_3 in EG **(d)** γ in DR

(e) δ in SC **(f)** ω in AD

Fig. 8: AUTOTRAINER Detection Precision and Recall vs. Configurable Parameters.

detection. The configurable parameters can be divided into three categories, as shown as follows.

Type-A: Type-A parameters include α_1, α_2, α_3, and α_4 in Table 1, which are used to determine the time window used to detect the occurrence of VG, EG, DR, and AD problems. For these problems, the same symptoms can also be observed in the rest of the training iterations once they occur in one iteration. We observe and confirm this phenomenon with 50 models and 3 runs on each model. This is also supported by the existing work [23]. Thus, we set α_1, α_2, α_3 and α_4 to 0.

Type-B: AUTOTRAINER has only one Type-B parameter, the expected accuracy threshold Θ, which is a training task-dependent parameter. If not, we will adopt the value that is used to determine if the training should be early stopped.

Type-C: Parameters in this category include β_1, β_2, and β_3 for VG and EG; γ for DR; δ and ζ for OL; η for SC; and ω for AD (defined in Table 1). The values of these parameters determine whether AUTOTRAINER can successfully capture the real problem or not. To measure their effects on AUTOTRAINER, for each problem, we use different values to investigate how they can affect the detection effectiveness. All experiments are performed on 100 models and repeated 5 times, and the final averaged results of the precision and recall are shown in Fig. 8.

• **VG**: The values of β_1 and β_2 affect the detection results of VG. If β_1 or β_2 is too large, it will introduce a lot of false positives (i.e., the benign training is considered to suffer from VG problem). If they are too small, it will reduce the detection accuracy (i.e., true positives), as shown in Fig. 8(a). The default values in AUTOTRAINER (i.e., β_1: $1e^{-3}$, β_2: $1e^{-4}$) can achieve high precision and recall.

• **EG**: Fig. 8(c) presents the relationship between precision/recall and the value of β_3. Larger β_3 will miss many EG cases that are less serious. AUTOTRAINER selects 70 as the default value of β_3 to achieve 100% precision and recall simultaneously, as shown in Fig. 8(c).

• **DR**: The detection results of DR is highly affected by the value of γ (see Fig. 8(d)). Larger γ can increase the precision but may ignore some DR problems that are not so severe. When γ is set in range $[60\%, 90\%]$, AUTOTRAINER achieves the optimal result and the default value of γ is set as 70%.

• **OL**: Detecting the OL problem requires ζ and η, and its relationships with precision and recall in detection are shown in Fig. 8(b). It is consistent with our intuition that larger ζ and η values will lead to higher precision and lower recall. In AUTOTRAINER, the default values for ζ and η are 0.03 and 20%, which results in 100% precision and recall.

• **SC**: δ is used to detect the SC problem. A very small δ results in very low precision and recall and too large δ values may result in the ignorance of many buggy training cases, leading to low precision. From Fig. 8(e), to achieve the optimal precision and recall, δ should be from 0.004 to 0.014. AUTOTRAINER selects the default value of δ as 0.01.

• **AD**: ω is the threshold to determine the AD problem during training. As shown in Fig. 8(f), when ω is too small, it will be difficult for AUTOTRAINER to detect AD problems effectively, resulting in lower precision. A large ω will cause a large number of false positives, resulting in a decrease in the recall. To ensure the detection effect of AUTOTRAINER, we set the default value of ω as 10.

Answer to RQ3: Configurable parameters can greatly affect the detection precision and recall of AUTOTRAINER, and improper parameters could lead to a large number of false positives and false negatives. Based on the large-scale evaluations, AUTOTRAINER selects a suitable set of configurable parameter values as default to obtain the optimal precision and recall simultaneously.

5.5 Effectiveness of Built-in Solutions

Experiment Design and Results: To evaluate the effectiveness of the built-in solutions in AUTOTRAINER, we apply each repair method corresponding to the problem to repair it separately and record the repair result and accuracy improvement. We repeated the experiment 3 times to eliminate the randomness, and Table 6 shows the averaged repair effect and accuracy improvement of each solution on each dataset. "Solve" and "Improv." in Table 6 represent the proportion of models repaired by the solution to all models to be repaired and the average absolute accuracy improvement of successful repairs, respectively. The subsequent columns show the effects of applying each solution on each dataset. The left-to-right order of repair methods in the table is the default order of solutions when AUTOTRAINER tries to repair a training problem.

Analysis: The experimental results demonstrate the effectiveness of all the built-in solutions for repairing training problems. For both VG and EG problems, the repair solutions can effectively solve the problem and achieve an accuracy improvement of up to 52.96%. The repair solutions for IO, IL, and AD problems have achieved a 100% repair

TABLE 6: The Repair Effects of the Built-in Solutions

Dataset	Repair Result (%)	VG		EG			DR			OL				SC			IO	IL	AD
		S2	*S1*	*S2*	*S1*	*S3*	*S2*	*S1*	*S4*	*S7*	*S6*	*S4*	*S5*	*S7*	*S6*	*S4*	*S2*	*S8*	*S9*
Blob	***Solve***	100.00	66.67	100.00	100.00	100.00	100.00	100.00	66.67	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
	Improv.	18.96	17.44	33.73	-7.33	-14.80	7.22	2.89	-12.89	-3.00	-2.33	-2.50	-27.33	41.44	30.10	-2.25	15.56	-0.04	16.27
Circle	***Solve***	87.50	75.00	100.00	100.00	100.00	100.00	100.00	0.00	100.00	100.00	100.00	100.00	97.67	90.70	2.33	100.00	100.00	100.00
	Improv.	21.29	20.71	85.33	49.42	54.87	13.48	16.00	-	9.27	8.60	2.00	-7.33	33.05	26.64	-0.02	6.90	10.11	24.37
CIFAR-10	***Solve***	100.00	100.00	100.00	100.00	100.00	100.00	100.00	0.00	100.00	100.00	100.00	100.00	96.43	96.43	17.86	100.00	100.00	100.00
	Improv.	56.74	58.59	52.36	0.50	1.67	12.34	58.85	-	10.97	8.03	4.42	-15.45	49.98	33.01	0.26	19.82	40.79	31.64
MNIST	***Solve***	100.00	100.00	50.00	100.00	100.00	100.00	100.00	50.00	100.00	100.00	100.00	100.00	100.00	100.00	40.00	100.00	100.00	100.00
	Improv.	87.64	87.37	0.02	14.61	51.60	86.02	87.84	0.00	56.82	56.55	17.99	-30.81	82.77	70.65	3.36	74.89	0.22	0.14
Reuters	***Solve***	0.00	100.00	100.00	0.00	100.00	-	-	-	0.00	100.00	100.00	100.00	95.00	100.00	35.00	100.00	100.00	-
	Improv.	-	18.59	52.07	-	0.85	-	-	-	-	19.75	5.63	1.96	40.24	31.08	0.02	36.57	16.16	-
IMDB	***Solve***	100.00	100.00	100.00	100.00	0.00	-	-	-	50.00	100.00	100.00	100.00	90.91	72.73	9.09	100.00	100.00	-
	Improv.	36.56	37.63	85.94	50.12	-	-	-	-	1.26	0.50	-8.27	-7.41	30.04	11.91	0.00	11.43	10.02	-
Total	***Solve***	91.18	85.29	92.11	84.21	86.84	100.00	100.00	23.53	76.47	100.00	100.00	100.00	97.32	94.63	32.89	100.00	100.00	100.00
	Improv.	36.64	37.58	52.96	18.45	18.24	29.24	38.15	-2.27	13.34	15.15	2.45	-13.23	45.17	33.88	0.09	22.20	11.22	18.39

success rate. Across all models on the six datasets, they improved the accuracy by 22.20%, 11.22%, and 18.39%, respectively. It is worth noting that the repair effect of the solutions on forward propagation problems is usually slightly worse than that of solutions on backward propagation training problems. For example, "S2" achieves an accuracy improvement of up to 52.96% on the EG problem but only 22.20% on the IO problem. The underlying reason for this phenomenon is consistent with our analysis in §5.2, i.e., when forward propagation problems occur, training tends not to come to a complete standstill (which is different from backward propagation problems like VG and EG), and the model can still update weights and improve accuracy. In addition, compared to other solutions, "S4" and "S5" have low repair success rates of 23.53% and 32.89% for DR and SC problems, indicating their limited applicability to problematic models. Consequently, these two solutions have the lowest priority in repairing the DR and OL problems. AUTOTRAINER priorities solutions that achieve better repair results (e.g., "S2" and "S1") to effectively and efficiently repair training problems in the model.

We further analyze the distribution and repair effects of training problems on different models and datasets and have the following observations. ❶ The occurrence of some training problems is related to model architecture and training data. For example, the DR problem does not occur on the RNN model that does not use the ReLU activation function. There is no AD problem in the models we collected using text datasets because ML frameworks (e.g., TensorFlow) provide serialization preprocessing APIs for text data. In addition, the VG and EG problems occur more frequently in models using the IMDB dataset. This may be related to the curse of dimensionality [61] of high-dimensional data, making training more difficult to converge and prone to these gradient problems. ❷ One solution could have different effects on different models and datasets. As shown in Table 6, "S2" can effectively solve VG and EG problems on various models, but fails to repair the VG problem on several models with the Reuters dataset. Our analysis shows that limited by the complexity of these models and their depth of up to 21 layers, replacing the activation function cannot effectively solve the VG problem. Gradient problems can still accumulate during backward propagation. However, "S1" can effectively amplify the gradients through the BN layer and alleviate the problems on these models. In addition, we can observe that "S1" cannot effectively solve the EG problem on the models with the Reuters dataset. This is because the complex model structure causes the gradient value to overflow at the beginning of the backward propagation, making normalization ineffective on these models. Moreover, "S7" obtained a poor repair effect on the text dataset (e.g., the Reuters dataset). This phenomenon may be related to the sparsity of the gradient of high-dimensional data, and substituting optimizers cannot help the model converge to the local minimum.

Answer to RQ4: All the solutions in AUTOTRAINER have the capability to repair the training problem, and most of them can fix the training problems with a high success rate and significantly improve the model accuracy. Furthermore, the repair effectiveness of solutions varies across models and datasets. To effectively repair the given training problem, the built-in repair methods of AUTOTRAINER will prioritize those with high overall success rates and good repair effects on various models.

6 THREATS TO VALIDITY

AUTOTRAINER has three limitations. First, the detection and repair performance on particular models may be degraded. We have tried our best to obtain as many models as possible. AUTOTRAINER is currently evaluated on 6 datasets and 701 models, which may be limited. Similarly, the effects of the configurable parameter setting in AUTOTRAINER may not hold when the number of models is significantly larger. Furthermore, although the 9 built-in solutions in AUTOTRAINER have proved their effectiveness in the prior work and our experiment, there is no guarantee that they will not introduce other severe training problems on some particular models, although this has never been encountered in our experiments.

Second, although AUTOTRAINER covers 8 common training problems, there are still some rare but severely affected problems that jeopardize the model training performance. These uncovered problems will result in the repaired model still having lower accuracy. We will continue to update the training problems supported by AUTOTRAINER and continue to maintain AUTOTRAINER to cover more problems and provide better guarantees for model training.

Third, AUTOTRAINER currently does not support the detection and repair of large models (LMs) (e.g., GPT [62], BERT [63]). On the one hand, the architecture and usage of LMs are different from that of CNNs or RNNs. The current mainstream deployment and use of LMs are mainly through downloading and finetuning pre-trained models.

Most users hardly need and do not have the resources to train the model from scratch, therefore, there are few reports of the training problem on LMs and very little need to repair the LMs' training. Even if the model faces problems in the pre-training [64], it is difficult to reproduce them due to hardware limitations. On the other hand, existing problems and research on LMs are mainly related to the underlying hardware and software [65, 66], e.g., parallel strategies on multiple GPU cores [67], which is different from the algorithmic-level model training problem of AUTOTRAINER detection and repair, and not in our scope of concern.

7 RELATED WORK

Machine learning techniques are widely adopted in various software engineering tasks. The closest related work in terms of design goals is DeepLocalize [27], which analyzes DNN's traces to identify and localize the faults in the models. In contrast, AUTOTRAINER aims to provide real-time detection and repair for various training problems in the DNN model. Our work is highly related to DNN model debugging and testing, automatic program repair, and automated machine learning.

7.1 DNN Model Debugging and Testing

In addition to what we have discussed in §1, there are some other efforts devoted to debugging DNN models [68, 69]. LAMP [70] utilized gradient information as data provenance to help debug graph-based machine learning algorithms. Ma et al. [5] proposed differential analysis on inputs to fix model overfitting and underfitting problems. Different from the existing debugging research, our work focuses on designing an automated and real-time detecting and repairing system for various training problems in the DNN model. Our work summarizes the symptoms of these training problems and evaluates the problem solutions to ensure timely and effective fixes.

A great number of testing methods have been proposed to test machine learning models, such as fuzzing [71], symbolic execution [72], fairness testing [73], etc. DeepXplore [74] introduced the neuron coverage metric to measure the percentage of activated neurons or a given test suite and DNN model. Model testing is useful for other domains such as image classification [72], automatic speech recognition [75], and text classification [76]. Yan et al. [77] have studied many coverage criteria and measured their correlations with model quality (i.e., model robustness against adversarial attacks), and empirical results show that existing criteria can not faithfully reflect model quality. Different from the above research, our work aims to monitor and test the DNN training problems in real-time.

7.2 Automatic Program Repair

The purpose of automatic program repair is to automatically derive patches to correct bugs in programs, usually including fault location, patch candidate generation, and patch candidate verification. Many different kinds of methods have been employed in automated program repair. The researchers have proposed search-based [78, 79], semantics-based [80, 81] and specifications-based methods [82, 83] to generate patches. Recently, Gissurarson et al. [84] leveraged both property-based testing and the rich type system and synthesis to design a novelty property-based automatic program repair method. More program repair work can be found in the survey [85]. Different from these research efforts, our work is to repair DNN models that are not uninterpretable rather than interpretable code.

7.3 Automated Machine Learning

Automated machine learning (AutoML) focuses on automatically designing and building models for given training tasks. Various kinds of AutoML algorithms and tools have been design to find efficient models [86–90]. However, models generated by AutoML may still encounter training problems during training. The goals of AutoML and AUTOTRAINER are different, and these two are complementary solutions that can be integrated with each other.

8 DISCUSSION

Insights for Improving DNN Quality: ❶ Although AUTOTRAINER can monitor and repair 8 training problems in real-time, it still faces 1.14 times extra time overhead due to repeated training and searching for suitable repair solutions. How to design more efficient and effective training problem detection and repair tools will be a direction of future research. Our experiment results show that solutions (e.g., "S7") that adjust training configurations can solve many problematic models. Studying and constructing suitable training configurations for different model architectures and downstream tasks (e.g., sentiment analysis and image classification) may be one of the directions for effectively solving training problems in the future. ❷ Except for the model and training configurations, training data also affects the training results and DNN model quality, leading to converging failure and even introducing other problems such as bias and backdoors [91, 92]. How to build an automated pipeline to debug and clean training data will be a future direction. ❸ The life cycle of DL models includes multiple stages such as training, inference, and fine-tuning, and the models could still encounter new problems outside the training stage that degrade the DNN quality, such as catastrophic forgetting [93]. Different from the training problem, problems in inference and fine-tuning stages cannot be solved by substituting model activation functions or layers, which seriously affects the trained parameters in the model. How to build a monitoring and repair framework for other stages of the model life cycle will be a future direction.

Software Bugs in Model Training: DL software bugs could cause model training to fail to converge, increase training overhead, or even crash [64, 94]. Even worse, such bugs cannot be repaired by the model-level solutions implemented in AUTOTRAINER. How to design an automated framework to detect and locate these possible software bugs during training is of great significance to the development of AI software technology and security.

9 CONCLUSION

This paper presents AUTOTRAINER, an automatic monitoring and repairing system for DNN model training. It

provides a real-time monitor for model training to identify potential problems according to the problem symptoms and automatically fixes them with built-in solutions. By doing so, it can prevent and fix training problems at an early stage, saving a lot of time and computation resources. Our experiment results show that AUTOTRAINER can effectively and efficiently detect and repair eight training problems (i.e., vanishing gradient, exploding gradient, dying ReLU, oscillating loss, slow convergence, improper output activation function, improper loss function, and abnormal data) and improve the average accuracy of 36.42% on six datasets.

REFERENCES

[1] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Autotrainer: An automatic dnn training problem detection and repair system," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 359–371.

[2] "Global edge ai software market," 2022, https://www.researchandmarkets.com/reports/5638910/global-edge-ai-software-market-by-component-by.

[3] "Google cloud, harvard global health institute release improved covid-19 public forecasts, share lessons learned," 2021, https://cloud.google.com/blog/products/ai-machine-learning/google-and-harvard-improve-covid-19-forecasts.

[4] "Providence creates chatbot to improve covid-19 care and manage hospital resources," 2021, https://customers.microsoft.com/en-in/story/1402681562485712847-providence-health-provider-azure-en-united-states.

[5] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2018, pp. 175–186.

[6] G. Cadamuro, R. Gilad-Bachrach, and X. Zhu, "Debugging machine learning models," in *ICML Workshop on Reliable Machine Learning in the Wild*, 2016.

[7] A. Chakarov, A. Nori, S. Rajamani, S. Sen, and D. Vijaykeerthy, "Debugging machine learning tasks," *arXiv preprint arXiv:1603.07292*, 2016.

[8] D. Mané *et al.*, "Tensorboard: Tensorflow's visualization toolkit, 2015."

[9] "Autotrainer repository," 2022, https://github.com/shiningrain/AUTOTRAINER.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[11] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," *arXiv preprint arXiv:1710.05941*, 2017.

[12] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[13] Y. Sun, X. Wang, and X. Tang, "Deeply learned face representations are sparse, selective, and robust," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2892–2900.

[14] N. Ketkar and E. Santana, *Deep Learning with Python*. Springer, 2017, vol. 1.

[15] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[16] C. Chuang and Y. Mroueh, "Fair mixup: Fairness via interpolation," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=DNl5s5BXeBn

[17] T. Pang, K. Xu, Y. Dong, C. Du, N. Chen, and J. Zhu, "Rethinking softmax cross-entropy loss for adversarial robustness," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020. [Online]. Available: https://openreview.net/forum?id=Byg9A24tvB

[18] "Pytorch documentations," 2020, https://pytorch.org/docs/stable/index.html.

[19] Y. LeCun, "The mnist database of handwritten digits," *http://yann.lecun.com/exdb/mnist/*, 1998.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[21] I. Arnekvist, J. F. Carvalho, D. Kragic, and J. A. Stork, "The effect of target normalization and momentum on dying relu," *arXiv preprint arXiv:2005.06195*, 2020.

[22] C. Xing, D. Arpit, C. Tsirigotis, and Y. Bengio, "A walk with sgd," *arXiv preprint arXiv:1802.08770*, 2018.

[23] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, no. 1, 1991.

[24] D. Sussillo and L. Abbott, "Random walk initialization for training very deep feedforward networks," *arXiv preprint arXiv:1412.6558*, 2014.

[25] J. Miller and M. Hardt, "Stable recurrent models," *arXiv preprint arXiv:1805.10369*, 2018.

[26] "Convolutional neural networks for visual recognition," 2020, https://cs231n.github.io/neural-networks-3/.

[27] M. Wardat, W. Le, and H. Rajan, "Deeplocalize: fault localization for deep neural networks," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 251–262.

[28] Y. Li, L. Meng, L. Chen, L. Yu, D. Wu, Y. Zhou, and B. Xu, "Training data debugging for the fairness of machine learning software," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2215–2227.

[29] K. Kim, B. Ji, D. Yoon, and S. Hwang, "Self-knowledge distillation with progressive refinement of targets," in *Proceedings of the IEEE/CVF international conference on computer vision*, 2021, pp. 6567–6576.

[30] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[31] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, "Understanding batch normalization," in *Advances in Neural Information Processing Systems*, 2018, pp. 7694–7705.

[32] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," in *Advances in neural information processing systems*, 2017, pp. 971–980.

[33] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, no. 1, 2013, p. 3.

[34] A. Graves, "Generating sequences with recurrent neural networks," *arXiv preprint arXiv:1308.0850*, 2013.

[35] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, 2013, pp. 1310–1318.

[36] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, "Dying relu and initialization: Theory and numerical examples," *arXiv preprint arXiv:1903.06733*, 2019.

[37] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[38] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[39] "How to control the stability of training neural networks with the batch size," 2019, https://machinelearningmastery.com/how-to-control-the-speed-and-stability-of-training-neural-networks-with-gradient-descent-batch-size.

[40] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *arXiv preprint arXiv:1804.07612*, 2018.

[41] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

[42] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[43] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*, 2016.

[44] N. Qian, "On the momentum term in gradient descent learning algorithms," *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[45] S. Krishnan and E. Wu, "Alphaclean: Automatic generation of data cleaning pipelines," *arXiv preprint arXiv:1904.11827*, 2019.

[46] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu, "Boostclean: Automated error detection and repair for machine learning," *arXiv preprint arXiv:1711.01299*, 2017.

[47] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[48] "Sklearn, make circles dataset," 2020, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_circles.html.

[49] "Sklearn, make blobs dataset," 2020, https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html.

[50] "Cifar-10 datasets," 2020, https://www.cs.toronto.edu/~kriz/cifar.html.

[51] "Imdb datasets," 2020, https://www.imdb.com/interfaces/.

[52] "Reuters-21578 dataset," 2020, http://www.daviddlewis.com/resources/testcollections/reuters21578/.

[53] "scikit-learn, machine learning in python," 2020, https://scikit-learn.org/stable/.

[54] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-Based Learning Applied to Document Recognition," *Proceedings of the IEEE*, 1998.

[55] "How to avoid exploding gradients with gradient clipping," 2019, https://machinelearningmastery.com/how-to-avoid-exploding-gradients-in-neural-networks-with-gradient-clipping.

[56] "The first layer's weights don't change after training when using keras," 2020, https://stackoverflow.com/questions/55874958/the-first-layers-weights-dont-change-after-training-when-using-keras.

[57] "How to avoid exploding gradients with gradient clipping," 2020, https://stackoverflow.com/questions/53848232/tensorflow-exploding-gradient.

[58] J. E. Pérez, J. Díaz, J. G. Martín, and B. Tabuenca, "Systematic literature reviews in software engineering - enhancement of the study selection process using cohen's kappa statistic," *J. Syst. Softw.*, vol. 168, p. 110657, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110657

[59] S. M. Vieira, U. Kaymak, and J. M. C. Sousa, "Cohen's kappa coefficient as a performance measure for feature selection," in *FUZZ-IEEE 2010, IEEE International Conference on Fuzzy Systems, Barcelona, Spain, 18-23 July, 2010, Proceedings*. IEEE, 2010, pp. 1–8. [Online]. Available: https://doi.org/10.1109/FUZZY.2010.5584447

[60] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

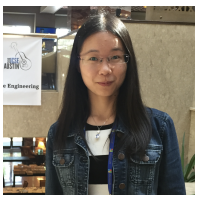
[61] N. Altman and M. Krzywinski, "The curse (s) of dimensionality," *Nat*

Methods, vol. 15, no. 6, pp. 399–400, 2018.
[62] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.
[63] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[64] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin *et al.*, "Opt: Open pre-trained transformer language models," *arXiv preprint arXiv:2205.01068*, 2022.
[65] "Large language models like gpt-3 have hardware problems," 2022, https://www.analyticsinsight.net/large-language-models-like-gpt-3-have-hardware-problems.
[66] "Overcoming the 4 major challenges in training openai models," 2023, https://www.spaceo.ai/blog/challenges-in-training-openai-models/.
[67] W. Fedus, B. Zoph, and N. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *The Journal of Machine Learning Research*, vol. 23, no. 1, pp. 5232–5270, 2022.
[68] G. Tao, S. Ma, Y. Liu, Q. Xu, and X. Zhang, "Trader: Trace divergence analysis and embedding regulation for debugging recurrent neural networks," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.
[69] X. Gao, J. Zhai, S. Ma, C. Shen, Y. Chen, and Q. Wang, "Fairneuron: Improving deep neural network fairness with adversary games on selective neurons," *arXiv preprint arXiv:2204.02567*, 2022.
[70] S. Ma, Y. Aafer, Z. Xu, W. Lee, J. Zhai, Y. Liu, and X. Zhang, "LAMP: data provenance for graph based machine learning algorithms through derivative computation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 786–797. [Online]. Available: https://doi.org/10.1145/3106237.3106291
[71] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: A coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.
[72] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 109–119.
[73] P. Zhang, J. Wang, J. Sun, G. Dong, X. Wang, X. Wang, J. S. Dong, and D. Ting, "White-box fairness testing through adversarial sampling," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.
[74] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.
[75] M. H. Asyrofi, Z. Yang, J. Shi, C. W. Quan, and D. Lo, "Can differential testing improve automatic speech recognition systems?" in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 674–678.
[76] W. Huang, Y. Sun, X. Zhao, J. Sharp, W. Ruan, J. Meng, and X. Huang, "Coverage-guided testing for recurrent neural networks," *IEEE Transactions on Reliability*, 2021.
[77] S. Yan, G. Tao, X. Liu, J. Zhai, S. Ma, L. Xu, and X. Zhang, "Correlations between deep neural network model coverage criteria and model quality," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 775–787. [Online]. Available: https://doi.org/10.1145/3368089.3409671
[78] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.
[79] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 254–265.
[80] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32rd International Conference on Software Engineering*. ACM, 2010, pp. 215–224.
[81] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.
[82] B. Demsky and M. Rinard, "Data structure repair using goal-directed reasoning," in *International Conference on Software Engineering, 2005. ICSE 2005. Proceedings.*, 2005, pp. 176–185.
[83] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 637–647.
[84] M. P. Gissurarson, L. Applis, A. Panichella, A. van Deursen, and D. Sands, "Propr: property-based automatic program repair," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1768–1780.
[85] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2017.
[86] H. Jin, Q. Song, and X. Hu, "Auto-keras: An efficient neural architecture search system," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 1946–1956.
[87] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," in *European Conference on Computer Vision*. Springer, 2020, pp. 544–560.
[88] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.
[89] X. Zhang, J. Zhai, S. Ma, and C. Shen, "Dream: Debugging and repairing automl pipelines," *arXiv preprint arXiv:2401.00379*, 2023.
[90] "Microsoft nni," 2020, https://github.com/microsoft/nni.
[91] R. Wang, P. Chaudhari, and C. Davatzikos, "Bias in machine learning models can be significantly mitigated by careful training: Evidence from neuroimaging studies," *Proceedings of the National Academy of Sciences*, vol. 120, no. 6, p. e2211613120, 2023.
[92] M. Xue, C. He, J. Wang, and W. Liu, "One-to-n & n-to-one: Two advanced backdoor attacks against deep learning models," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1562–1578, 2020.
[93] X. Li, Y. Zhou, T. Wu, R. Socher, and C. Xiong, "Learn to grow: A continual structure learning framework for overcoming catastrophic forgetting," in *International conference on machine learning*. PMLR, 2019, pp. 3925–3934.
[94] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

Xiaoyu Zhang received the B.E. degree in Automation from Xi'an Jiaotong University, China in 2020. He is currently working toward the Ph.D. degree in Xi'an Jiaotong University, China. His current research interests include AI security and software engineering.

Chao Shen received the B.E. degree in Automation from Xi'an Jiaotong University, China in 2007; and the Ph.D. degree in Control Theory and Control Engineering from Xi'an Jiaotong University, China in 2014. He is currently a Professor in the Faculty of Electronic and Information Engineering, Xi'an Jiaotong University of China. His current research interests include AI Security, insider/intrusion detection, behavioral biometrics, and measurement and experimental methodology.

Shiqing Ma received the Ph.D. degree in Computer Science from Purdue University in 2019. He is an Assistant Professor in the Manning College of Information & Computer Sciences, University of Massachusetts, Amherst, United States. His research focuses on improving the transparency of modern computing systems, which is an intersection of security, AI, and software systems.



Juan Zhai is an Assistant Professor with the Manning College of Information & Computer Sciences, University of Massachusetts, Amherst, United States. Her main research interests include software engineering and program languages, natural language processing and software text analytics, software reliability and security, deep learning.

Chenhao Lin received the B.E. degree in automation from Xi'an Jiaotong University in 2011, the M.Sc. degree in electrical engineering from Columbia University, in 2013 and the Ph.D. degree from The Hong Kong Polytechnic University, in 2018. He is currently a Research Fellow in Xi'an Jiaotong University of China. His research interests are in artificial intelligence security, adversarial attack and robustness and interpretable machine learning.