AUTOTRAINER: An Automatic DNN Training Problem Detection and Repair System

Xiaoyu Zhang*, Juan Zhai†, Shiqing Ma†, Chao Shen*
*School of Cyber Science and Engineering, Xi'an Jiaotong University, Xi'an, China
†Rutgers University, United States
Email: zxy0927@stu.xjtu.edu.cn, {juan.zhai, shiqing.ma}@rutgers.edu, chaoshen@xjtu.edu.cn

***Abstract*—With machine learning models especially Deep Neural Network (DNN) models becoming an integral part of the new intelligent software, new tools to support their engineering process are in high demand. Existing DNN debugging tools are either post-training which wastes a lot of time training a buggy model and requires expertises, or limited on collecting training logs without analyzing the problem not even fixing them. In this paper, we propose AUTOTRAINER, a DNN training monitoring and automatic repairing tool which supports detecting and auto-repairing five commonly seen training problems. During training, it periodically checks the training status and detects potential problems. Once a problem is found, AUTOTRAINER tries to fix it by using built-in state-of-the-art solutions. It supports various model structures and input data types, such as Convolutional Neural Networks (CNNs) for image and Recurrent Neural Networks (RNNs) for texts. Our evaluation on 6 datasets, 495 models show that AUTOTRAINER can effectively detect all potential problems with 100% detection rate and no false positives. Among all models with problems, it can fix 97.33% of them, increasing the accuracy by 47.08% on average.**

***Index Terms*—software engineering, software tools, deep learning training**

I. INTRODUCTION

In the new Software Engineering (SE) 2.0 era, software is developed with an intelligent component which is usually powered by Machine Learning (ML) techniques. Recent advances in Deep Learning (DL) have already made it possible for end users to benefit from the intelligence of software. For example, Google has deployed new DL based NLP techniques to help improve its search results [1]. Facebook launched Shops which bring more businesses online during the COVID pandemic, and made it possible to search for clothes using over tens of thousands of image attributes. All of these are enabled by software created in SE 2.0.

With this new trend of SE 2.0, developing the DL component, represented by Deep Neural Network (DNN) models, becomes an integral part of the whole process. DNN models and other DL methods, just like other programs, also have bugs and its own vulnerabilities, which brings many new challenges of SE research on debugging and repairing DNN model and its development process. New tools that can help the development of intelligent components will greatly help developers especially for the ones who are new to these techniques. There are already some efforts trying to study this problem [2, 3]. For example, MODE [4], proposed as a DNN debugging technique, identifies faulty neurons that lead to undesirable behaviors and selects additional training samples to correct these neurons behaviors to improve model accuracy. We refer to such techniques as *post-training* techniques, which focuses on fixing model problems whose training has been completed. However, many existing tools are not automatic and require expertises, which makes them difficult to use for developers new to this field. More importantly, we observe that *many DNN problems have been exposed in the training process*, and post-training techniques have a delay in detecting such problems. As such, a lot of resources are wasted in training a problematic model which can be saved if we can detect the problem early in training. Thus, a runtime monitoring and detecting technique is highly needed.

Existing DNN training frameworks have provided limited support for training monitoring and detection. TensorBoard [5], known as the default debugger of TensorFlow is a toolkit which can record various values and provide the visualization during the training process. For example, it can track and visualize metrics like loss, and demonstrate histograms of weights as they change over time. There are some other similar tools, such as Visdom [6], TensorWatch [7], and Manifold [8]. Just like traditional debuggers (e.g., Microsoft Visual Studio Debugger) which allow programmers to track variable values and operations as well as monitoring the changes of computing resources, these tools can facilitate developers in inspecting and understanding the model training status. However, they lack the capability of analyzing the collected data and provide meaningful fixes, which makes them less useful.

Through our analysis, we found that 1) a training problem happens or not is random even for the same training script; 2) a training problem happens randomly during the training whole. To be more specific, because of the randomness in training (e.g., initialization of weight values, training data sequences), running the same training scripts may get different results. As such, a training problem may happen in some cases but not in other cases. And similarly, a training problem may happen in any training iteration (if it happens). We have provided real cases in §III. Considering the fact that many DNN training tasks may take days or even months, it is infeasible for developers to watch the numbers or curves all the time to manually detect potential problems which may occur at any time. Unfortunately, although these runtime tools can collect and exhibit data, they are incapable of analyzing the data to diagnose problems, let alone leveraging solutions to alleviate

these training problems. To address the aforementioned limitations, a tool that relieves developers from manually monitoring the training procedure is needed. In addition, the tool will have to automatically analyze data, diagnose and resolve problems during training, so as to increase productivity and efficiency for developers as well as improve the reliability of intelligent software systems.

In this paper, we propose AUTOTRAINER, a dynamic approach that detects and repairs potential DNN training problems. The training problems that AUTOTRAINER focuses on are vanishing gradient, exploding gradient, dying ReLU, oscillating loss and slow convergence, and AUTOTRAINER is capable of handling various model structures including convolutional neural networks (CNNs) and Recurrent Neural Networks (RNNs). And these can be easily extended as long as a problem definition is provided. Given a model with its training configuration (e.g., hyper-parameter, optimizers), AUTOTRAINER will start training the model and record relevant data like loss values. During the monitoring, AUTOTRAINER conducts regular analysis to recognize potential training problems. If a problem is detected, AUTOTRAINER will try to fix it with built-in solutions. These solutions are constructed based on the state-of-the-art work, which have been demonstrated to work well in solving the corresponding problems [9–12]. During the repair (retraining) procedure, if another problem is detected, AUTOTRAINER will regard the old problem as resolved and attempt to repair the new problem. If no more problems are detected, it means all the problems have been addressed by AUTOTRAINER and the trained model with its configuration is delivered to the user. If AUTOTRAINER fails to solve this problem, it will notify the user with complete training log. Our contributions are:

- We summarize and formalize definitions for the symptoms of 5 common training problems.
- We propose the first automatic approach to detect and repair 5 different training problems during model training.
- We develop a prototype AUTOTRAINER based on the proposed idea, and evaluate it with 6 public datasets and 495 models. The evaluation results demonstrate that AUTOTRAINER can effectively detect all 316 problems for 262 models and repair 309 problems of them with a ratio of 97.78%. On average, the test accuracy can be improved from 32.46% to 79.54% (1.5x higher).
- Our implementation, collected datasets, configurations, and problem solutions are publicly available at [13].

Threat to Validity. We have tried our best to obtain as many models as possible. AUTOTRAINER is currently evaluated on 6 datasets and 495 models, which may still be limited. Similarly, there are many configurable parameters used in AUTOTRAINER, and even though our experiments show that they are good enough to achieve high detection and repair results, this may not hold when the number of models is significantly larger. To mitigate these threats, all the original and repaired training scripts, model architecture and training configuration details, implementation including dependencies, and evaluation data (e.g., training logs) are publicly available at [13] for reproduction.

II. BACKGROUND

A. 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. It usually composes of several connected layers. Formally, an n-layered DNN can be written as $F = l_1 \circ l_2 \circ \cdots \circ l_n$, where l represent a layer. Each l can be expressed as a function whose output is $F_l = \sigma(\theta_l * F_{l-1} + b_l)$ where θ_l and b are the weight and bias values of layer l. σ is known as the activation function (§II-C). The input layer l_1 takes raw inputs and passes them on to the subsequent layer. 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 links between consecutive layers are represented using a set of matrices. The numerical values in such matrices are referred to as weight parameters. Given a large set of input-output pairs (x_i, y_i), training a DNN model is to update all weight parameters θ to minimize the differences between a predicted result $F_\theta(x)$ and the corresponding ground truth label y. Such differences are measured by a loss function $\mathcal{L}(F_\theta(x), y)$. Thus, training a DNN essentially is to minimize the value of $\mathcal{L}$.

Specifically, training a DNN model consists of the following phrases. The first step is *initialization* which initializes the weight matrices. Then starting from the input layer, the *forward propagation* step uses existing weight values to predict output labels for the training samples, and calculates the value of $\mathcal{L}$ based on predicted output and ground truth labels. Afterwards, the *backward propagation* step tweaks the weight values from the output layer all the way back to the input layer, trying to minimize the difference using an optimization method which is usually a gradient descent algorithm or its variants. The forward propagation and backward propagation steps will be repeated until the difference converges to a minimum value meaning reaching the stopping criteria, or has reached the maximal number of training iterations allowed.

B. Gradient Descent

In DNN model training, a loss function evaluates the prediction ability of a DNN model, and a smaller value of the loss function means a better model. Thus, the training goal is to obtain weight values which result in a minimum loss value. Gradient descent algorithm and its variants are commonly used to solve this optimization problem. It works by tweaking the weights in the opposite direction to the gradient of the loss function. Specifically, each weight has an update proportional to the partial derivative of the loss function with respect to the current weight. The gradients are usually calculated by auto differentiation (AD) techniques leveraging the chain rule. As such, computing the gradient for a weight has an effect of multiplying many numbers (from subsequent layers).

Normally, a neural network is designed to have many layers to improve its capacity. Increasing the number of layers can

enable a neural network to train on a large-scale training dataset and efficiently learn more complex mapping functions from inputs to outputs. However, the addition of layers can have negative impacts on training. The common problems are *vanishing gradient* and *exploding gradient*.

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

Symptoms of VG. The gradient decreases exponentially from layer to layer and is close to zero in the layers close to the input layer, and the training accuracy remain low.

Problem 2 (Exploding Gradient Problem)**.** *In contrast to VG, the gradient can grow exponentially as it is propagated backwards. This also leads to NaN or unexpected large values, which results in bad model accuracy. Such a problem is referred to as exploding gradient (EG).*

Symptoms of EG. The gradient increases exponentially from (output) layer to (input) layer during backward propagation and can become large or even *NaN* value in the layers close to the input layer, and the training accuracy is low.

C. Activation Function

Intuitively, each neuron in a DNN can be regarded as one special feature to differentiate between the given inputs. Given a set of inputs, each neuron computes the weighted sum and then adds a bias to the sum. After that, an activation function takes the computed sum as input and produces an output for the neuron. Specifically, the activation function determines how much the input is relevant for the following stage, guiding the neural network to leverage important features and suppress irrelevant features.

ReLU (Rectified Linear Unit) is a widely-used activation function in a neural network [14–16], which outputs the same value if the input is positive and outputs zero if the input is non-positive (i.e., $ReLU(x) = max\{x, 0\}$). Existing work [17] has demonstrated its excellent training effect. It effectively improves the sparsity of the model, achieving better training convergence and accuracy. However, using ReLU has its own limitations, among which *dying ReLU* is the most common and serious one.

Problem 3 (Dying ReLU)**.** *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 for an inactive neuron. Consequently, such neurons cannot be leveraged to distinguish between the inputs and ground truth, and if there are many such neurons, we may end up with a large part of the neural network contributing nothing to the prediction task. This is known as the dying ReLU (DR) problem.*

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

D. Convergence

The training goal is to reduce the loss value converge to a minimum. To determine the point of convergence, there are usually two conditions. One is that the training time has reached the maximal allowed iteration (defined by the user). And the other one is that the training accuracy has reached desired values. In some training cases, we may end up with a set of low accuracy models even after the maximal number of training iterations, and they are usually caused by two problems: oscillating loss and slow convergence.

Problem 4 (Oscillating Loss)**.** *It is inevitable for the loss value to go up and down during the training procedure. But if there are large changes without decreasing trend, the training may not converge in a very long time which should be enough for training the model. We refer to such a problem as oscillating loss (OL).*

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

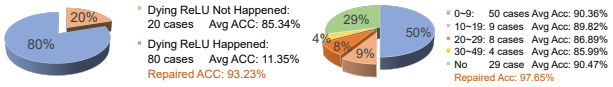
Problem 5 (Slow Convergence)**.** *The loss value has a high value and decrease so slow that no significant accuracy improvement has been made, and it may end up with low accuracy even when the maximal number of training iteration is finished. We refer to such a problem slow convergence (SC).*

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

III. Identifying DNN Problems During Training

As far as we know, there is no existing tool that can help users identify the aforementioned DNN problems during training. TensorFlow provides a TensorBoard Debugger [5] tool to help users inspect program variables (e.g., loss value) and inserting assertions. PyTorch also allows users to do the same thing by using PyTorch Hooks [18]. However, it requires expertises to perform the required analysis and patching to solve this problem. While many of these problems are common problems in DNN training, their symptoms and solutions have been studied and analyzed. In this paper, we propose AutoTrainer, a DNN training tool that can automatically monitor DNN internal values (i.e., neuron activations 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 automatically fix it. AutoTrainer is designed to be a training time monitoring and fixing tool because of the following:

- **Training problem occurrence is highly random.** When training a model using the same configuration and training dataset for multiple times, whether a problem occurs or not



(a) Occurrence **(b)** Time of Occurrence

Figure 1. Problems Occurrence and Time of Occurrence are Random

in a training procedure is random. This is because there are many random values used in DNN training. For example, the weight values are usually initialized with random values. In some cases, one problem will occur because of these random values while it will not happen in some other cases.

We train a DNN model with 34 layers (650,000 parameters) on the MNIST [19] handwritten digit dataset (50,000 training and 10,000 testing samples). In this model, we also use ReLU as the activation function, Adam as our optimizer, and set our learning rate to be 0.001 and the maximal number of epoch to 50 (see [13]). We train the model for 100 times. Figure 1(a) shows the distribution of the appearance of the dying ReLU problem. We can see that the dying ReLU problem occurs in 80% of the training processes but not in the remaining 20%, which demonstrates that whether a problem occurs or not is random. The average training accuracy when the DR problem happens is only 11.35% while the value reaches 85.34% when there is no DR problem. With AUTOTRAINER, we are able to detect all these DR problems and fix them, improving the accuracy to 93.23%.

• **The time when a training problem occurs is random.** Similar to the randomness of problem occurrence challenge, the time when the problem actually happens is also random during training. We use a model which has the oscillating loss problem as an example. It is a DNN model with 20 layers and uses ReLU as activation function, Adam as optimizer, and we set the learning rate to be 0.001 and the maximal number of iterations is 50. Details of the training scripts is also available in our repository [13]. The distribution of the stages (epoch number) when the problem occurs is shown in Figure 1(b). In 29% of the cases, the oscillating loss problem is not triggered. In half cases, the problem is detected in the first 10 epochs, and the percentages of detecting the problem in other stages are separately 9%, 8% and 4%. It demonstrates at which stage a particular problem occurs is random.

Since our system enforces real-time surveillance, it is able to perform timely detection and repair. For this model, AUTOTRAINER can detect the problem at 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). After the detection, our system attempts to resolve the problem by leveraging four solutions (i.e., substituting initializer, increasing batch size, decreasing learning rate and substituting optimizer. See §IV). Based on our experiments, the problem is successfully alleviated in all cases, and improves the accuracy to 97.65%. In contrast, existing *post-training* methods do not collect real-time data, making them unable to detect problems during the training.

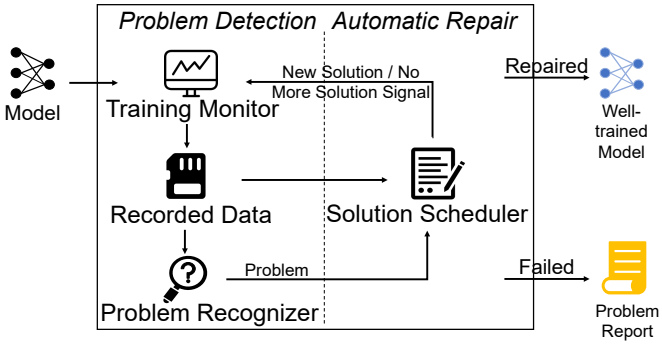


Figure 2. Overarching Design of AUTOTRAINER

IV. SYSTEM DESIGN

Figure 2 gives the overarching design of our system, which consists of the *problem detection* module (left) and the *automatic repair* module (right). The whole system starts by training a model with an initial training configuration and using the problem recognizer to monitor the training. When a problem is detected, the system will launch the automatic repair module trying to retrain the model with new settings until the training can finish without any problem (repaired) or a detected problem cannot be solved (failed). Notice that if there exist several problems, our system will attempt to solve the detected problems one by one in the order of exposure.

AUTOTRAINER takes a training configurations (i.e., the original training scripts including model architecture, loss function, optimizer etc.), and user preferences as inputs. The user preferences are configurable parameters for AUTOTRAINER, which includes preferred repair solutions and so on. AUTOTRAINER has a set of default values for them, and user can replace them. Details will be presented in §IV-B. The problem detector monitors training information like loss value. During this, the problem recognizer is triggered on a timely basis to analyze the recorded data to recognize symptoms and determine whether a training problem exists. If a problem is detected, the automatic repair module will be leveraged to address it. Otherwise, the training monitor will output the trained model with its training configurations to the user.

For each problem, AUTOTRAINER has a few built-in solutions to fix them. However, one solution may or may not work. If the detected problem is the same as before (if any), 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, the solution generator will select the corresponding solutions to it. The order of solutions can 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.

A. 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 includes:

- Model definition including layers and their configurations (e.g., kernel sizes in convolutional layers).
- Optimization method definition and its parameters.
- Training accuracy and loss values.
- Calculated gradients for each neuron.
- Hyper-parameters and other necessary variables used in training, such as the batch size and learning rate.

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

B. Problem Recognizer

The problem recognizer regularly conducts analysis on the recorded data to recognize training problems. The symptoms leveraged to detect problems are formalized and shown in Table I. The first column lists the training problems and the second column specifies the symptoms involving gradient and training accuracy. If the depicted condition is met, our system regards the corresponding symptom as observed. The last column presents the built-in solutions in AUTOTRAINER.

VG. We formalize the symptom of the VG problem 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 have terminated. Secondly, in the recent α_1 training iterations, the gradient has been drop from layer to layer in the backward propagation and the gradient becomes to be 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 [20–22].

EG. The definition of EG symptoms are very similar to that of VG except that the gradient is growing from layer to layer in backward propagation or it has already become NaN values in some layer (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 a few iterations ($[k - \alpha_3, k]$) and this set is large forms a large portion of the whole DNN (more than a threshold value γ) while the accuracy of the neuron net work is still low.

OL. Intuitively, the symptom of an OL problem is that there has 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 representing 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 oscillation is larger than η, we think this is a significant oscillation, and if such oscillation happens very frequently, we think there is an OL problem.

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

C. 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. And 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.

D. Existing Solutions

There has been some study on how to solve training problems. Unfortunately, there is no silver bullet and one solution cannot be guaranteed to work for all cases. For each problem, AUTOTRAINER collects a few possible solutions which have been shown to be effective in prior study and uses them to fix detected training problems, and these solutions include:

• **S1: Adding Batch Normalization Layers.** Batch normalization is a method used to normalize the neuron values of a layer by re-centering and re-scaling them. This helps remove the unexpected gradient and neuron activation values. Specifically, the normalization will squeeze the values into a specific range, and as such, small gradient updates will not diminish or explode during the backward propagation, meaning that the vanishing and exploding gradient problem can be alleviated [9, 26]. In addition, such value range enforcement reduces the possibility of getting an inactive neuron and help resolve the Dying ReLU problem.

Regarding to the problem of where to add batch normalization layers, authors of this method [9] has performed analysis and demonstrated that adding batch normalization before activation function layers gives the best result. We follow this guidance and implemented our solution.

• **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 (if used with the proper optimizer and learning rate). Hence, substituting the current activation function with ReLU and its variants (e.g., SELU [10], LeakyReLU [27]) can mitigate both the vanishing gradient problem and the exploding gradient problem.

• **S3: Adding Gradient Clipping.** Gradient clipping clips gradient values that exceed a specified range, which essentially limits the update of a weight value to a limited region. Unlike batch normalization and other normalization methods, this method clips the gradient values based on a threshold. By removing obviously large gradient values, it can be used to alleviate the exploding gradient problem [28–30].

Bengio et al. [30] and many others [29] have studied and evaluated the concrete values to use in gradient clipping, and

TABLE I: Problem Symptoms and Repair Solution Candidates

Training Problem	Symptom	Solution
Vanishing Gradient [20–22]	Gradient: $\forall i \in [k-\alpha_1, k], \frac{\left\|G^i_{l_2}\right\|}{\left\|G^i_{l_3}\right\|} \ldots \frac{\left\|G^i_{l_{n-1}}\right\|}{\left\|G^i_{l_n}\right\|} \leq \beta_1 \bigwedge \left\|G^i_{l_2}\right\| \leq \beta_2$ Accuracy: $max(Acc) \leq \Theta$	S1: Adding Batch Normalization Layers S2: Substituting Activation Functions
Exploding Gradient [20–22]	Gradient: $\forall i \in [k-\alpha_2, k], \frac{\left\|G^i_{l_2}\right\|}{\left\|G^i_{l_3}\right\|} \ldots \frac{\left\|G^i_{l_{n-1}}\right\|}{\left\|G^i_{l_n}\right\|} \geq \beta_3 \bigvee \exists j \in N, G^i_j = NaN$ Accuracy: $max(Acc) \leq \Theta$	S1: Adding Batch Normalization Layers S2: Substituting Activation Functions S3: Adding Gradient Clip
Dying ReLU [23]	Gradient: $\forall i \in [k-\alpha_3, k], \frac{\lvert\{j \in N \lvert, G^i_j = 0\}\rvert}{\lvert N \rvert} \geq \gamma$ Accuracy: $max(Acc) \leq \Theta$	S1: Adding Batch Normalization Layers S2: Substituting Activation Functions S4: Substituting Initializer
Oscillating Loss [24]	Accuracy: $\frac{\lvert\{i \in [1, min(\lvert A \rvert, \lvert B \rvert)] \lvert A[i] - B[i] \geq \zeta\}\rvert}{k} \geq \eta$	S4: Substituting Initializer S5: Adjusting Batch Sizes S6: Adjusting Learning Rate S7: Substituting Optimizer
Slow Convergence [25]	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

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, arraies of maximum/minimum optimal

recommend to clip the gradient of each layer to $[-10, 10]$ which is adopted by AUTOTRAINER.

• **S4: Substituting Initializers.** Initializers set a starting point for the optimizers during model training. Thus, inappropriate initialization can cause disasters in training deep neural networks. Xavier initialization [31] was proposed solve the oscillating loss and slow convergence problem by initiating the weight values to a proper range. Lu et al. [11] have shown that the popular initialization schemes like He Initialization [32] suffers from the Dying ReLU problem. In §III, we also showed a case where the initialization values can affect the model training. Thus, we also try 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 error gradient, which is an important hyper-parameter. A too large batch size might make the loss value trap into a poor local minimum, leading to low accuracy, while a too small batch size might make the loss bounce around a lot, leading to oscillating loss [12, 33]. In practice, increasing the batch size appropriately has the potential to improve the stability of the training and solve the oscillating loss problem.

Setting the proper batch size is not easy for DNN training, and various researchers have performed analysis and evaluation and this [12, 29, 30]. Based on our study, LeCun et al. and Bengio et al. suggest using 32 as the starting point. Following this, we try to initialize the batch size to be 32. If the accuracy is too low, we try to descries the batch size by a factor of 2 until it reaches 8. If the OL occurs, we will increase the batch size also by doubling until it reaches 256 (according to Masters et al. [12]).

• **S6: Adjusting Learning Rates.** Learning rate is a hyper-parameter that 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 fluctuating update and the loss value will oscillate and even increase over training epochs [28]. Given this, decreasing the learning rate can be helpful to tackle the oscillating loss problem.

Generally, a small learning rate makes it possible for 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. As such, the slow convergence problem might be resolved if the learning rate is increased [28].

The values we set for learning rates depend on different optimizers they use. We follow the suggestions made by their original authors (e.g., Adam [34]) and existing empirical evidence [35, 36]. 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 have different performance in different scenarios. Practically, a substitution of an optimizer can help address various training problems. Stochastic Gradient Descent (SGD) [37] is a variant of the basic gradient decent algorithm, which 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. However, the high variance in weights may result in fluctuations in the loss value.

Momentum [38] is a method introduced to speed up SGD and dampen loss oscillations. It works by adding a fraction of the update in the past time step to the current update.

Usually the value of momentum is set as 0.9 or a similar value. The value 0.9 means the weights will update based on 90% of the previous gradient and 10% of the new gradient. Such a mechanism achieves a faster convergence and fewer oscillations compared with SGD. However, if the momentum is too much, we may swing back and forth near the local minimum without hitting the minimum. Adaptive Moment Estimation (Adam) [34] is a widely adopted optimizer that uses momentum and adaptive learning rates. The adaptive learning rate allows us to start with large learning rate and finish with small learning rate. As the learning rate is decreasing, we will take smaller and smaller steps, which can prevent us from missing the local minimum and accelerate the convergence.

In a nutshell, we can use algorithms with momentum like SGD+Momentum or Adam to alleviate the oscillating loss problem, and randomly use a different optimizer to alleviate the slow convergence problem.

V. Evaluation

The prototype of AutoTrainer is implemented on top of Keras 2.3.1 [39] and TensorFlow 2.1.0 [40]. In the evaluation, we aim to answer the following research questions:

RQ1: How effective is AutoTrainer in detecting and fixing training problems?

RQ2: How efficient is AutoTrainer in detecting and fixing training problems?

RQ3: What is the impact of different configurable parameters in AutoTrainer?

A. Setup

We performed our experiments on six popular datasets: Circle [41], Blob [42], MNIST [19], CIFAR-10 [43], IMDB [44] and Reuters [45]. Circle and Blob are two datasets from SKLearn [46] for classification tasks. MNIST is a gray-scale image dataset used for handwritten digits recognition. CIFAR-10 is a colored image dataset used for object recognition. IMDB is a movie review dataset for sentiment analysis. Reuters is a newswire dataset for document classification. In total, we collected 495 models and their training scripts with various DNN models structures (CNN, RNN and fully connected layers only) for these six datasets. Among them, 262 of them have training problems and the rest are normal models, which have been confirmed by model authors. Most models are collected from reported buggy models on GitHub, StackOverflow, existing papers and personal blogs, and some of them are gathered from machine learning experts within our organization. The training scripts of these models and all experiment results are all public at our repository [13].

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 a NVIDIA TITAN V GPU running Ubuntu 18.04 as the operating system.

B. Effectiveness of AutoTrainer

Experiment Design: To evaluate the effectiveness of AutoTrainer, we run our collected 495 model training scripts to test the effectiveness of AutoTrainer. Due to the randomness in performing these experiments, we run the training multiple times to ensure that the problem have been exposed, and collect the experiment results of such cases. To measure the effectiveness of AutoTrainer, we start two parallel trainings for the same model. To reduce the effects of randomness, we enforce them to 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 collected training logs including gradient, loss values, etc., to help us verify the whole process. At the end, we manually verify them and confirm them with the details provided by buggy model providers (i.e., online posts and machine learning experts).

Results: For all 262 buggy trainings, we detect 316 training problems as some models have more than one. Table II demonstrates partial results. The first column lists the six datasets. The second column shows the model status and the corresponding number of models. "Repaired" status indicates a model is successfully repaired if any target problem is detected and "Failed" indicates that the problem still exists even after AutoTrainer has tried all built-in solutions. Lastly, we use the "Normal" status to denote models without training problems. The third column lists model identification numbers, and the fourth column shows the number of detected problems of each model. The following columns denote the accuracy, the training time and the memory consumption (efficiency results, see §V-C). Column "Original" shows the information for the original model training (without AutoTrainer) while column "AT" (short for AutoTrainer) shows that of the repaired models. Column "Ratio" gives the ratio between the values of a repaired model and the corresponding original model. Column "Improve" shows the absolute accuracy improvement 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 least accuracy improvement, minimum training overhead and the minimum memory overhead. All detailed experiment results can be found at our repository [13].

Notice that the same problem may get repaired using different solutions, and after AutoTrainer repairs the model, its accuracy may not be improved. To evaluate the repair effects, we also calculated the number of problems that are fixed by individual solutions and the change ranges in accuracy. The results are separately shown in Table III, Table IV and Figure 3. In Table III, the first column shows the datasets, and the following columns present the problem and corresponding solutions. The solutions are (from left to right) ordered by 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

TABLE II: Overall Results of AUTOTRAINER

Dataset	Status	Model	#Problem	Accuracy				Train Time			Average Memory		
				Original (%)	*AT (%)*	*Improve (%)*	*Ratio*	*Original (s)*	*AT (s)*	*Ratio*	*Original (MB)*	*AT (MB)*	*Ratio*
Blob	Repaired: 46	1	1	20.33	86.00	65.67	4.23	18.53	44.35	2.39	1580.68	1571.75	0.99
		2	1	85.33	84.67	-0.67	0.99	31.06	65.06	2.09	1554.12	1552.32	1.00
		3	3	30.67	85.00	54.33	2.77	15.86	586.63	36.98	1564.95	1565.12	1.00
		4	1	40.33	83.67	43.33	2.07	17.99	24.03	1.34	1550.91	1554.17	1.00
		5	1	53.00	84.67	31.67	1.60	21.00	33.04	1.57	1575.69	1582.19	1.00
		6	1	50.33	83.67	33.33	1.66	12.99	30.99	2.39	1282.06	1258.63	0.98
		Ave	1.13	43.31	81.04	37.73	1.87	13.67	51.78	3.79	1505.91	1505.24	1.00
	Failed: 2	47	1	33.67	33.67	0.00	1.00	16.03	109.56	6.84	1576.94	1574.19	1.00
		48	1	33.67	33.67	0.00	1.00	6.89	86.64	12.57	1574.70	1573.95	1.00
		Ave	1	33.67	33.67	0.00	1.00	11.46	98.10	8.56	1575.82	1574.07	1.00
	Normal: 39	49	-	67.67	-	-	-	15.61	17.74	1.14	1505.91	1486.04	0.99
		50	-	70.67	-	-	-	15.71	16.94	1.08	1514.26	1509.71	1.00
		Ave	-	76.27	-	-	-	16.36	17.68	1.08	1562.39	1561.12	1.00
Circle	Repaired: 71	88	1	n.a	88.33	88.33	n.a	18.00	28.44	1.58	1320.03	1321.28	1.00
		89	1	87.33	86.33	-1.00	0.99	22.98	71.85	3.13	1325.53	1326.22	1.00
		90	2	49.67	78.00	28.33	1.57	3.50	57.18	16.35	1332.55	1332.32	1.00
		91	1	54.33	87.67	33.33	1.61	60.88	74.27	1.22	1358.89	1358.89	1.00
		Ave	1.10	46.97	83.56	36.60	1.78	28.05	67.60	2.41	1336.85	1336.12	1.00
	Normal: 36	159	-	76.67	-	-	-	16.20	16.94	1.05	1305.43	1302.58	1.00
		160	-	68.00	-	-	-	29.14	30.53	1.05	1313.31	1306.58	0.99
		Ave	-	81.43	-	-	-	28.01	29.29	1.05	1342.73	1342.25	1.00
CIFAR-10	Repaired: 45	195	2	10.00	71.73	61.73	7.17	73.48	1180.53	16.07	4477.29	3774.16	0.84
		196	1	57.72	69.46	11.74	1.20	73.80	139.98	1.90	4359.05	3655.91	0.84
		197	2	10.00	65.67	55.67	6.57	409.89	431.13	1.05	3021.97	3004.28	0.99
		198	1	8.12	65.44	57.32	8.06	93.24	129.31	1.39	3744.07	5034.64	1.34
		199	1	10.00	67.82	57.82	6.78	214.44	295.44	1.38	3589.40	2817.81	0.79
		Ave	1.02	13.11	66.80	53.69	5.10	248.96	522.22	2.10	3679.23	3513.32	0.95
	Failed: 1	240	1	10.00	10.00	0.00	1.00	382.79	492.60	1.29	3777.13	3777.13	1.00
	Normal: 35	241	-	65.37	-	-	-	342.88	319.32	0.93	4452.32	3749.12	0.84
		242	-	56.56	-	-	-	236.45	244.89	1.04	3749.78	3753.79	1.00
		Ave	-	63.48	-	-	-	145.63	146.39	1.01	3946.77	3826.03	0.97
MNIST	Repaired: 38	276	1	9.33	99.17	89.84	10.63	148.82	212.36	1.43	3084.37	3083.77	1.00
		277	1	88.44	99.12	10.68	1.12	267.11	401.37	1.50	3226.07	3188.81	0.99
		278	2	9.80	99.20	89.40	10.12	365.55	2294.69	6.28	3283.42	3283.67	1.00
		279	1	35.08	98.89	63.81	2.82	331.44	365.42	1.10	3475.25	3286.04	0.95
		280	1	9.50	99.21	89.71	10.44	123.55	225.97	1.83	3143.98	3356.33	1.07
		281	1	16.14	99.11	82.97	6.14	65.40	123.88	1.89	3347.91	3138.69	0.94
		Ave	1.13	16.22	98.88	82.66	6.10	220.74	493.14	2.23	3085.09	3026.66	0.98
	Normal: 78	314	-	98.79	-	-	-	173.85	173.47	1.00	3203.96	3198.19	1.00
		315	-	86.54	-	-	-	135.93	136.42	1.00	3168.69	2925.68	0.92
		Ave	-	96.89	-	-	-	228.43	228.79	1.00	3251.75	3203.40	0.99
Reuters	Repaired: 31	392	1	4.19	66.74	62.56	15.93	645.01	1023.98	1.59	2247.90	2361.28	1.05
		393	1	56.86	57.30	0.45	1.01	855.34	2844.38	3.33	2413.77	2481.97	1.03
		394	1	47.91	52.45	4.54	1.09	1278.41	4778.25	3.74	2312.99	2360.14	1.02
		395	1	n.a.	62.91	62.91	n.a.	1316.38	1321.00	1.00	1859.90	1869.22	1.01
		396	1	n.a.	58.46	58.46	n.a.	2591.11	4363.19	1.68	1823.66	1758.23	0.96
		Ave	1	21.37	59.23	37.87	2.77	1411.38	2894.73	2.05	2214.09	2250.93	1.02
	Failed: 1	423	1	36.02	36.02	0.00	1.00	1484.00	1460.37	0.98	1881.61	1820.14	0.97
	Normal: 32	424	-	47.13	-	-	-	1466.05	1510.32	1.03	1912.03	1927.08	1.01
		425	-	46.15	-	-	-	1400.59	1471.97	1.05	1901.38	1930.53	1.02
		Ave	-	51.75	-	-	-	1222.65	1249.35	1.02	2307.96	2347.96	1.02
IMDB	Repaired: 24	456	1	n.a	87.08	87.08	n.a	3982.20	8876.97	2.23	1998.46	2069.23	1.04
		457	1	82.04	80.53	-1.51	0.98	2078.05	5961.56	2.87	2251.86	2285.11	1.01
		458	1	49.33	86.03	36.70	1.74	670.69	6925.99	10.33	2256.18	2342.93	1.04
		459	1	49.12	85.95	36.83	1.75	1069.43	1307.58	1.22	2258.64	2345.51	1.04
		460	1	49.78	86.68	36.90	1.74	1306.62	5478.55	4.19	2138.05	2317.73	1.08
		461	1	50.00	86.11	36.11	1.72	3703.41	9246.12	2.50	2057.98	1854.65	0.90
		Ave	1	45.29	84.39	39.10	1.86	2871.01	6499.83	2.26	2153.72	2195.81	1.02
	Failed: 3	480	1	50.00	50.00	0.00	1.00	1062.77	1480.36	1.39	2182.29	2349.58	1.08
		481	1	n.a	n.a	0.00	n.a	2089.21	1451.98	0.69	2184.26	2044.23	0.94
		482	1	n.a	n.a	0.00	n.a	1300.46	2257.56	1.74	2261.88	2348.53	1.04
		Ave	1	16.67	16.67	0.00	1.00	1484.15	1729.97	1.17	2209.47	2247.45	1.02
	Normal: 13	483	-	87.38	-	-	-	1951.87	1984.37	1.02	2260.38	2347.84	1.04
		484	-	83.14	-	-	-	2022.07	2036.59	1.01	2191.92	2088.60	0.95
		Ave	-	84.54	-	-	-	1828.85	1844.67	1.01	2224.19	2280.91	1.03
Normal		Ave	-	79.14	-	-	-	375.37	380.57	1.01	2591.48	2565.53	0.99
Repaired		Ave	1.07	32.46	79.54	47.08	2.45	515.66	1141.79	2.21	2224.98	2195.30	0.99
Failed		Ave	1	23.34	23.34	0.00	1.00	906.02	1048.44	1.16	2205.54	2212.54	1.00

TABLE III: The Problem Repaired Results

Dataset	VG		EG			DR			SC			OL				Total
	S2	*S1*	*S2*	*S1*	*S3*	*S2*	*S1*	*S4*	*S7*	*S6*	*S4*	*S7*	*S6*	*S5*	*S4*	
Blob	10	2	10	0	0	4	3	1	29	0	0	4	0	0	0	63
Circle	9	1	9	1	0	6	3	0	43	1	0	7	1	0	0	81
CIFAR-10	5	0	7	1	0	2	1	0	27	1	0	2	0	0	0	46
MNIST	6	2	10	0	0	4	0	0	20	1	0	7	1	0	0	51
Reuters	0	3	6	0	0	-	-	-	19	7	0	0	4	0	0	39
IMDB	5	2	5	0	1	-	-	-	9	3	0	0	4	0	0	29
Total	35	10	47	2	1	16	7	1	147	13	0	20	10	0	0	309
Repaired		45			50			24			160				30	309
Failed		3			4			0			0				0	7
Total		48			54			24			160				30	316

TABLE IV: The Accuracy Improvement of Problems

Dataset	#Repaired					Avg Improve(%)				
	VG	*EG*	*DR*	*SC*	*OL*	*VG*	*EG*	*DR*	*SC*	*OL*
Blob	12	10	8	29	4	38.78	30.40	24.78	43.01	-0.50
Circle	10	10	9	44	8	30.08	82.92	28.10	33.64	10.20
CIFAR-10	5	8	3	28	2	56.17	58.50	59.81	52.72	11.74
MNIST	8	10	4	21	8	87.72	87.68	86.74	82.93	56.97
Reuters	3	6	-	26	4	20.12	50.30	-	37.58	20.77
IMDB	7	6	-	12	4	36.43	85.94	-	33.43	-0.55
Total/Average	45	50	24	160	30	45.04	66.87	46.91	46.05	16.00

solution (i.e., column name). The bottom half summaries the number of problems of different status. The pie chart in Figure 3 demonstrates the distrubution of change ranges in accuracy with corresponding numbers of models.

Analysis. The experiment results demonstrate the effectiveness of our system. Firstly, AUTOTRAINER can effectively detect the defined training problems with a 100% success rate on all 495 model trainings, and none of the normal trainings are mis-classified as problematic. Secondly, AUTOTRAINER can effectively repair the buggy training procedures. After successful repairing, it can improve the accuracy by 47.08% with the maximum improvement as 89.84%. Overall, AUTOTRAINER achieves around 46% accuracy improvement which is 2.43 times that of training without AUTOTRAINER. Notice that when EG happens, it may result in NaN values in the model, 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 II

From Table III, we observe that AUTOTRAINER is able to respectively repair 93.75%, 92.6%, 100%, 100% and 100% of VG, EG, DR, SC and OL in buggy trainings. And Table IV shows the detailed average accuracy improvement of AUTOTRAINER on different problems, which is 45.04%, 66.87%, 46.91%, 46.05% and 16.00% of VG, EG, DR, SC and OL in repairing, respectively. Table III and Table IV demonstrate that AUTOTRAINER is capable of handling different types of datasets, model problems and model architectures. Regarding SC and OL, we find that only two solutions can effectively address all the problems we encountered in our evaluation.

Figure 3 shows that model accuracy increase distrubution. Specifically, over 40.78% of the models get an increase of 50% and over 50% has an increase between 10% and 30%. It demonstrates that AUTOTRAINER has the advantage of effectively increasing accuracy by repairing training problems. We also notice that there are 6 models (out of all 255 models)

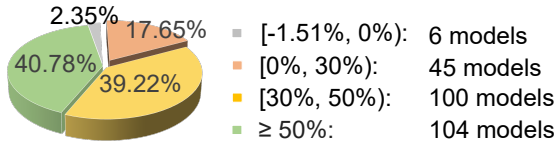
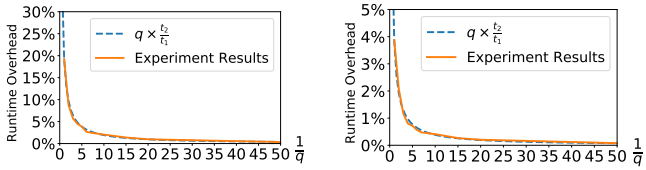


Figure 3. Accuracy Change After Model Training Repair.



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

Figure 4. Runtime Overhead vs. Problem Check Frequency.

whose accuracy has slight reduction after repaired. For the worst case, the model accuracy decreases from 82.04% to 80.53%. We manually analyzed them, and found that it is mainly because they have other problems that are not covered by AUTOTRAINER, such as floating point bugs. How to detect and repair such problems will be one of our future directions.

C. *Efficiency of* AUTOTRAINER

Experiment Design and Results: To evaluate the efficiency of AUTOTRAINER, we run all 495 model trainings with and without AUTOTRAINER enabled. During training, we collected the time used to train the model and the memory usage for both the original training and AUTOTRAINER. Notice that experiments for run-time overhead and memory overhead are conducted individually to avoid influencing each other. The experiments are conducted 5 times, and the overhead is calculated as the average of these 5 runs. The results are shown in Table II. Results and analysis are presented below.

Runtime Overhead Analysis: For normal training, the runtime overhead is purely from problem checker, which is about 1%. From Table II, 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. For buggy trainings, AUTOTRAINER takes 1.19 more training time on average. We performed a deeper analysis to understand the overhead of individual components, and found that retraining takes over 99% and the rest two parts (i.e., problem checker and repair) takes less than 1%. It means that AUTOTRAINER only costs little time ($\leq 1\%$ total overhead) in automatically searching a suitable solution for the problems, which requires lots of manual operations and time-consuming in existing strategies. As discussed in §IV, to repair a problem, it may try several times, which leads AUTOTRAINER training several models.

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

TABLE V: 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

$q \times t_2/t_1$, where q is the checking frequency. Figure 4 presents the corelations between checking frequency and runtime overhead on Circle and MNIST. 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, shapes of experiment data conform to our theoretical analysis. By comparing the two figures, we observe the smaller dataset has the higher runtime overhead, which is consistent with our data in Table II. By default, AUTOTRAINER checks the problem every 3 iterations, which causes less than 5% overhead even for small datasets like Circle.

Lower frequency checking can reduce the overhead, but may cause longer delay in detection. Table V shows the effects of different check frequencies. 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 the other iteration. In a nutshell, a lower checking frequency may result in the delay in problem detection, which further leads to wasting time on a buggy training.

Memory Overhead Analysis. AUTOTRAINER has very limited memory overhead (-1% to 1%) since AUTOTRAINER reuses data which has been collected. To detect problems, AUTOTRAINER requires the current gradient values, and historical loss and training accuracy values. The gradient information is stored as part of the tensor data for training purpose (used in backward propagation), and the historical loss and training accuracy values are automatically collected by all major frameworks. The overhead caused by AUTOTRAINER are mostly due to program variables, which are negligible compared with neuron and gradient values and even the overhead introduced by the memory profiling tool itself.

D. Effects of Configurable Parameters

AUTOTRAINER leverages configurable parameters to determine if a problem is happening or not. We classify them into three categories and evaluated each of them in this section.

Type-A: Type-A parameters include α_1, α_2 and α_3 in Table I, which are used to determine the time window used to detect the occurrence of VG, EG and DR problems. For these problems, the same symptoms will also be observed in the rest iterations once they happen in one iteration. We comfirmed this with 50 models and 3 runs on each model. This is also supported by others [20]. Thus, we set α_1, α_2 and α_3 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 which is used to determine if the training should be early stopped, and it is provided by Keras and TensorFlow.

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; and η for SC (defined in Table I). 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 (or value pairs if a problem involves multiple such parameters) to investigate how they can affect the detection effectiveness. All experiments are performed on 100 models, and they are repeated 5 times. Figure 5 reports the final averaged results.

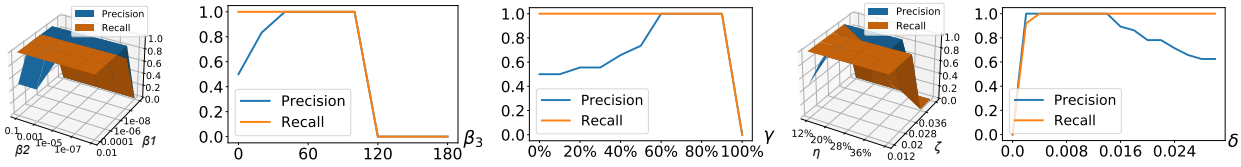
• **VG**: The values of β_1 and β_2 affect the detection results of VG. If β_1 or β_2 is too large, it will introduce a lot of false positives (i.e., normal trainings are identified as VG). If they are too small, it will reduce the detection accuracy (i.e., true positives). Figure 5(a) demonstrates how precision and recall are changed as the parameter values change. It confirms the aforementioned analysis and suggests the default values in AUTOTRAINER (i.e., β_1: $1e^{-3}$, β_2: $1e^{-4}$) can achieve high precision and recall.

• **EG**: Figure 5(b) presents the relationship between precision/recall and the value of β_3. Larger β_3 implies that EG becomes more obvious, but it also means many EG cases which are less serious will be missed. Hence, precision gets higher but recall becomes very low. Luckily, when β_3 is between 40 and 100, AUTOTRAINER can get 100% precision and recall simultaneously. By default, AUTOTRAINER sets β_3 to 70.

• **DR**: The detection results of DR is highly affected by the value of γ (see Figure 5(c)). With larger γ, AUTOTRAINER is able to remove obvious False Positive (FP) cases (i.e., detected as DR, but is not DR) increasing the precision, but also may ignore not so serious DR problem causing low recall. When γ is set in range $[60\%, 90\%]$, AUTOTRAINER achieves the best result in precision (100%) and recall (100%). By default, γ is set as 70% in AUTOTRAINER.

• **OL**: Detecting OL requires two parameters, ζ and η, and its relationships with precision and recall in detection are shown in Figure 5(d). 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 100% recall.

• **SC**: The only threshold in detecting SC is δ. Very small δ results in very low precision and recall. On one hand, such low accuracy change are not common during training (hence, low recall), and on the other hand, many of them with such low accuracy change are due to randomly initialized weights (hence, low precision). With no-so-large δ values, the detection precision and recall can grow sharply. However, larger δ values may result in the ignorance of many buggy training cases, leading to low precision. Figure 5(e) shows such a change



(a) Vanishing Gradient **(b)** Exploding Gradient **(c)** Dying ReLU **(d)** Oscillating Loss **(e)** Slow Convergence

Figure 5. AUTOTRAINER Detection Precision and Recall vs. Configurable Parameters.

curve. Based on our sampling, to achieve the best in precision (100%) and recall(100%), δ should be from 0.004 to 0.014. In AUTOTRAINER, we set the default value of δ as 0.01.

VI. RELATED WORK

Machine learning techniques are widely adopted in various SE tasks [47–61]. AUTOTRAINER can facilitate software engineering researchers in repairing their buggy DNN models automatically. It is highly related with DNN model debugging and testing, and automatic program repair.

DNN Model Debugging and Testing. In addition to what we have discussed in §I, there are some other efforts devoted on debugging DNN models [62–66]. Ribeiro et al. [62] produced adversarial examples as training data to debug natural language processing models. Others [63, 64] cleaned up training data that are wrongly labeled to debug RNN models. LAMP [67] utilizes gradient information as data provenance to help debug graph based machine learning algorithms. Ma et al. [4] proposed differential analysis on inputs to fix model overfitting and under fitting problems. TRADER [65] analyzed how problematic word embeddings affect the model accuracy by comparing the model execution traces of correctly-classified samples and incorrectly-classified samples.

A great number of testing methods have been proposed to test machine learning models, such as fuzzing [68–75], symbolic execution [76–79], runtime validation [80, 81], fairness testing [74, 78, 82], etc. DeepXplore [83] introduced the neuron coverage metric to measures the percentage of activated neurons or a given test suite and DNN model, and generates new test inputs that can maximize the metric to test DL systems. Many others [70, 84–88] extended the coverage concept and proposed to use them on many different scenarios. Model testing has also been leveraged for many other domains such as image classification [79, 89], automatic speech recognition [90], text classification [74], and machine translation [91, 92]. Recently, Yan et al. [93] 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.

Automatic Program Repair. The aim of automatic program repair is to automatically derive patches to correct bugs in programs, which normally includes fault localization, patch candidates generation and patch candidates validation. Many different kinds of methods have been employed in automated program repair. The researchers in [94–99, 99] proposed search-based approaches to generate patches. There are some other semantics-based methods which construct patches using synthesis techniques [100, 100–104]. Specifications were also utilized to guide the repair process [105–109]. More program repair work can be found in the survey [110]. Different from these research efforts, our work is to repair DNN models which are not uninterpretable rather than the interpretable code.

Automated Machine Learning (AutoML). AutoML focuses on automatically design models for given training tasks. Various kinds of neural architecture search (NAS) algorithms have been design to find efficient models, such as Bayesian optimization [111, 112], deep reinforcement learning [113, 114], evolutionary algorithms [115, 116], and gradient based methods [117, 118]. These methods acquire impressive results in their experiments. Additionally, open-source AutoML tools, such as AutoSKLearn [119–121], Microsoft NNI [122], and AutoKeras [123], also show remarkable model searching results in actual application.

Although AutoML can automatically generate models from the training tasks, these models may still face training problems when training. Comparing with AutoML, AUTOTRAINER focuses on improving the training process. It can provide timely monitoring facing the training process and facilitates SE researchers in repairing buggy models automatically. In summary, the goal of AutoML and AUTOTRAINER are different, and these two are complementary solutions which can be intergraded with each other.

VII. CONCLUSION

This paper presents AUTOTRAINER, a DNN monitoring and auto-repairing system. It monitors the model training status and automatically fix them once a problem is detected. By doing so, it can prevent problems from happening at the earliest convention which saves a lot of time and resources. Our evaluation results show that AUTOTRAINER can effectively and efficiently detecting and repairing our targeted five training problems (i.e., vanishing gradient, exploding gradient, dying ReLU, oscillating loss and slow convergence).

ACKNOWLEDGEMENT

The authors thank the anonymous reviewers for their insightful feedback and constructive comments. We also thank Jiong Li for his efforts and feedbacks on this project. This work is, in part supported by the National Science Foundation of China (No. 61802166, 61822309, 61773310, U1736205) and National Key R&D Program of China under Grand No. 2020AAA0107700. Chao Shen is the corresponding author. The views, opinions and/or findings expressed are only those of the authors.

REFERENCES

[1] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *arXiv preprint arXiv:1909.11942*, 2019.

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

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

[4] 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.

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

[6] "Visdom," 2020, https://github.com/facebookresearch/visdom.

[7] "Tensorwatch," 2020, https://github.com/microsoft/tensorwatch.

[8] "Manifold," 2020, https://github.com/uber/manifold.

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

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

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

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

[13] "Autotrainer repository," 2020, https://github.com/shiningrain/AUTOTRAINER.

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

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

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

[17] 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.

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

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

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

[23] 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.

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

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

[26] 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.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] "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.

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

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

[36] 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.

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

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

[39] "Keras: The python deep learning library," 2020, https://keras.io.

[40] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

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

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

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

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

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

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

[47] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 612–621.

[48] D. Alrajeh and A. Russo, "Logic-based learning: Theory and application," in *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, 2018.

[49] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," *arXiv preprint arXiv:1901.09102*, 2019.

[50] C. S. Păsăreanu, D. Gopinath, and H. Yu, "Compositional verification for autonomous systems with deep learning components," in *Safe, Autonomous and Intelligent Vehicles*. Springer, 2019, pp. 187–197.

[51] R. Ben Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing advanced driver assistance systems using multi-objective search and neural networks," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 63–74.

[52] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering*, 2018.

[53] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 373–384.

[54] C. Chen, Z. Xing, and Y. Liu, "By the community & for the community: a deep learning approach to assist collaborative editing in q&a sites," *Proceedings of the ACM on Human-Computer Interaction*, 2017.

[55] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, M. Lanza, and R. Oliveto, "Sentiment analysis for software engineering: How far can we go?" in *Proceedings of 40th International Conference on Software Engineering (ICSE)*, 2018.

[56] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[57] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[58] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair," *arXiv preprint arXiv:1711.07163*, 2017.

[59] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 60–70.

[60] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016.

[61] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[62] M. T. Ribeiro, S. Singh, and C. Guestrin, "Semantically equivalent adversarial rules for debugging nlp models," in *Association for Computational Linguistics (ACL)*, 2018.

[63] X. Zhang, X. Zhu, and S. Wright, "Training set debugging using trusted items," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[64] Y. Jiang and Z.-H. Zhou, "Editing training data for knn classifiers with neural network ensemble," in *International symposium on neural networks*, 2004.

[65] 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.

[66] Y. Tian, S. Ma, M. Wen, Y. Liu, S. Cheung, and X. Zhang, "Testing deep learning models for image analysis using object-relevant metamorphic relations," *CoRR*, vol. abs/1909.03824, 2019. [Online]. Available: http://arxiv.org/abs/1909.03824

[67] 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

[68] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*, 2019, pp. 4901–4911.

[69] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: Differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 739–743.

[70] 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.

[71] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2018.

[72] J. Uesato, A. Kumar, C. Szepesvari, T. Erez, A. Ruderman, K. Anderson, N. Heess, P. Kohli *et al.*, "Rigorous agent evaluation: An adversarial approach to uncover catastrophic failures," *arXiv preprint arXiv:1812.01647*, 2018.

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

[74] S. Udeshi, P. Arora, and S. Chattopadhyay, "Automated directed fairness testing," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 98–108.

[75] X. Gao, R. Saha, M. Prasad, and R. Abhik, "Fuzz testing based data augmentation to improve robustness of deep neural networks," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[76] A. Ramanathan, L. L. Pullum, F. Hussain, D. Chakrabarty, and S. K. Jha, "Integrating symbolic and statistical methods for testing intelligent systems: Applications to machine learning and computer vision," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 786–791.

[77] D. Gopinath, C. S. Pasareanu, K. Wang, M. Zhang, and S. Khurshid, "Symbolic execution for attribution and attack synthesis in neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 282–283.

[78] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 625–635.

[79] 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.

[80] A. Stocco, M. Weiss, M. Calzana, and P. Tonella, "Misbehaviour prediction for autonomous driving systems," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

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

[82] 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.

[83] 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.

[84] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 120–131.

[85] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 303–314.

[86] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 132–142.

[87] H. Zhou, W. Li, Z. Kong, J. Guo, Y. Zhang, L. Zhang, B. Yu, and C. Liu, "Deepbillboard: Systematic physical-world testing of autonomous driving systems," in *2020 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[88] S. Gerasimou, H. F. Eniser, A. Sen, and A. Cakan, "Importance-driven deep learning system testing," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[89] Y. Tian, Z. Zhong, V. Ordonez, G. Kaiser, and B. Ray, "Testing dnn image classifier for confusion & bias errors," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[90] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 477–487.

[91] P. He, C. Meister, and Z. Su, "Structure-invariant testing for machine translation," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[92] Z. Sun, J. M. Zhang, M. Harman, M. Papadakis, and L. Zhang, "Automatic testing and improvement of machine translation," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.

[93] 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

[94] 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.

[95] W. Weimer, S. Forrest, C. L. Goues, and T. V. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the Acm*, vol. 53, no. 5, pp. 109–116, 2010.

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

[97] C. L. Goues, T. V. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[98] A. Arcuri, "Evolutionary repair of faulty software," *Applied Soft Computing*, vol. 11, no. 4, pp. 3494–3514, 2011.

[99] 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.

[100] 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.

[101] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 772–781.

[102] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *Ieee/acm IEEE International Conference on Software Engineering*, 2015, pp. 448–458.

[103] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 12–23.

[104] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 166–178.

[105] 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.

[106] B. Demsky, M. D. Ernst, P. J. Guo, S. McCamant, J. H. Perkins, and M. Rinard, "Inference and enforcement of data structure consistency specifications," in *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006, pp. 233–244.

[107] D. Gopinath, M. Z. Malik, and S. Khurshid, "Specification-based program repair using sat," in *International Conference on TOOLS & Algorithms for the Construction & Analysis of Systems*, 2011, pp. 173–188.

[108] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[109] 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.

[110] 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.

[111] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," *Advances in neural information processing systems*, vol. 25, pp. 2951–2959, 2012.

[112] 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.

[113] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[114] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[115] 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.

[116] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," *arXiv preprint arXiv:1703.01041*, 2017.

[117] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," in *Advances in neural information processing systems*, 2018, pp. 7816–7827.

[118] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *arXiv preprint arXiv:1812.00332*, 2018.

[119] "Auto-sklearn," 2020, https://github.com/automl/auto-sklearn.

[120] M. Feurer, A. Klein, K. Eggensperger, J. T. Springenberg, M. Blum, and F. Hutter, "Auto-sklearn: efficient and robust automated machine learning," in *Automated Machine Learning*. Springer, Cham, 2019, pp. 113–134.

[121] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, and F. Hutter, "Auto-sklearn 2.0: The next generation," *arXiv preprint arXiv:2007.04074*, 2020.

[122] "Microsoft nni," 2020, https://github.com/microsoft/nni.

[123] "Autokeras," 2020, https://github.com/keras-team/autokeras.