Hindawi
Security and Communication Networks
Volume 2023, Article ID 5968168, 14 pages
https://doi.org/10.1155/2023/5968168

*Research Article*

# Fed-DNN-Debugger: Automatically Debugging Deep Neural Network Models in Federated Learning

**Shaoming Duan**,[1,2,3] **Chuanyi Liu**,[1,2,3,4] **Peiyi Han**,[1,2,3,4] **Xiaopeng Jin**,[5] **Xinyi Zhang**,[6] **Xiayu Xiang**,[4] **and Hezhong Pan**[1]

[1]*School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen 518055, China*
[2]*Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies, Harbin Institute of Technology, Shenzhen 518055, China*
[3]*Shenzhen Key Laboratory of Data Security, Harbin Institute of Technology, Shenzhen 518055, China*
[4]*Peng Cheng Laboratory, Shenzhen 518000, China*
[5]*College of Big Data and Internet, Shenzhen Technology University, Shenzhen 518118, China*
[6]*School of Computer Science and Technology, The Chinese University of Hong Kong, Shenzhen 518172, China*

Correspondence should be addressed to Chuanyi Liu; liuchuanyi@hit.edu.cn and Peiyi Han; hanpeiyi@hit.edu.cn

Received 26 August 2022; Revised 14 January 2023; Accepted 17 January 2023; Published 23 February 2023

Academic Editor: Xuehu Yan



Federated learning is a distributed machine learning framework that has been widely applied in scenarios that require data privacy. To obtain a neural network model that performs well, when the model falls into a bug, existing solutions retrain it on a larger training dataset or the carefully selected samples from model diagnosis. To overcome this challenge, this paper presents Fed-DNN-Debugger, which can automatically and efficiently fix DNN models in federated learning. Fed-DNN-Debugger fixes the federated model by fixing each client model. Fed-DNN-Debugger consists of two modules for debugging a client model: nonintrusive metadata capture (NIMC) and automated neural network model debugging (ANNMD). NIMC collects the metadata with deep learning software syntax automatically. It does not insert any code for metadata collection into modeling scripts. ANNMD scores samples according to metadata and searches for high-quality samples. Models are retrained on the selected samples to repair their weights. Our experiments with popular federated models show that Fed-DNN-Debugger can improve the test accuracy by 8% by automatically fixing models.

## 1. Introduction

Federated learning [1–3] is a distributed machine learning framework in which multiple clients jointly train a model under the coordination of a central server. Each client can keep its data locally without disclosing them to the server or to other clients. Federated learning can protect data privacy and mine valuable information from data at the same time. As the most commonly used base model, deep learning models under the framework of federated learning have been widely applied in electronic health records [4, 5], traffic flow prediction [6, 7], medical image [8, 9], and other fields.

A deep learning model is a deep neural network (DNN) [10] with neurons connected by a set of weights, whose weights are updated by the training process. If a deep learning model performs poorly caused by the misconducted training processes (e.g., insufficient training or bias in the training dataset), it means that some weights of the model are not trained well [11, 12]. We call these weights as *error weights* in this paper. This can significantly degrade model

performance. For example, accuracy can be decreased by 10% because of this problem [11].

In federated learning, the training process and training data of each client cannot be accessed. The training data are collected by different parties, some of which have problems such as heterogeneity, noise, and unlabeled or mislabeled. Due to the lack of a global view of the data, it is difficult for data scientists to avoid the error weight problem in federated learning [13]. Therefore, fixing models in the federated learning is of great significance. A series of solutions have been proposed for centralized deep learning models. One solution to solve the problem of error weights is to add more training samples and modify the error weights by retraining. Collecting more samples, such as those from generative models for data augmentation [14–16], or from public data sources [17, 18], has limited effectiveness and sometimes even leads to degenerated models [11]. Another solution is to explore the relationship between error weights and training samples and provide guidance for selecting retraining samples that can repair error weights [11, 12, 19]. These approaches are more effective than the method of randomly increasing training samples. Unfortunately, they cannot be applied to federated learning because all of these methods were designed for centralized learning and rely on manual extraction and analysis of metadata (e.g., features or weights) generated during training. On the one hand, limited by privacy protection strategies, local training data and metadata cannot be accessed in federated learning. On the other hand, existing methods still impose significant computational and communication overhead in federated learning.

In this paper, we propose Fed-DNN-Debugger, a debugging system that can automatically and efficiently fix error weights in the federated model. Fed-DNN-Debugger consists of two modules: nonintrusive metadata capture (NIMC) and automated neural network model debugging (ANNMD). NIMC analyzes the data flows of client models with deep learning software syntax to automatically collect metadata, such as model features and model weights. It does not insert any code related to data collection and analysis into modeling scripts and is therefore nonintrusive. ANNMD automatically searches for high-quality samples with collected metadata. It efficiently fixes each client model by retraining models with selected samples.

We evaluate Fed-DNN-Debugger using four popular federated models on the MNIST and CIFAR-10 datasets. Experimental results show that Fed-DNN-Debugger can efficiently collect and filter out 99% of useless metadata. It takes only 4.3% of the execution time required by a native method that collects all metadata. Moreover, Fed-DNN-Debugger can automatically and effectively fix federated models. In particular, the accuracy can be increased by more than 8% for federated models. Even if only one client model is fixed, the test accuracy can increase by up to 5.75%.

The main contributions of our paper are as follows:

(1) We propose Fed-DNN-Debugger, a debugging system that can debug error weights of DNN models to improve their performance in federated learning

(2) We provide a nonintrusive method of metadata capture to efficiently collect training metadata from client models, without inserting any metadata collection code into deep learning scripts

(3) We propose an efficient method for automatic neural network debugging, which can search for high-quality samples and retrain models to repair error weights

(4) We develop a prototype of the Fed-DNN-Debugger and comprehensively evaluate it on popular federated models

## 2. Related Work

In this section, we introduce some work that is related to our research. Debugging methods for deep learning models are presented first because they are considered the basis of this paper. Furthermore, we discuss existing methods for model debugging in federated learning, after which we briefly introduce the difference between these studies and the research presented in this paper. Finally, we discuss how to capture the metadata during training.

*2.1. Deep Learning Model Debugging.* To debug deep learning models and improve their performance, various debugging methods have been proposed. Human-in-the-loop debugging [20] is the most commonly used method in the modeling process. Modelers use the data collection APIs provided by auxiliary debugging tools [21–23] to record and track the metadata during training. The main metadata to be collected include model hyperparameters, evaluation metrics (values of loss and accuracy), training samples, and models. According to the training metadata, the hyperparameters and structure of the model are manually analyzed and modified, or a generative adversarial network (GAN) is used for data augmentation [24–26]. Human-in-the-loop debugging relies mainly on the experience of the modeler and requires the modeler to analyze the data.

To repair the error weights of a model, Ma et al. [11] proposed an automatic neural network debugging method called MODE, inspired by software engineering debugging. It was powered by feature differential analysis and input selection to help identify buggy neurons and measure their importance for guiding the selection of new input samples. Retraining used high-quality samples to fix the model. Apricot [12] is a debugging method for weight adaptation during the training process. However, MODE and Apricot cannot be directly applied to federated learning.

*2.2. Federated Learning Model Debugging.* There are currently few works on federated learning model debugging. To the best of our knowledge, FLDebugger [27] is the only work for model debugging under federated learning framework. FLDebugger proposed a model debugging system for the problem of erroneous training data in federated learning. FLDebugger traces the test errors of the global model and the training log of each client during federated training process.

Then, FLDebugger fixes model bugs by identifying erroneous training samples based on influential functions [28, 29]. Unlike FLDebugger, which solves the problem of erroneous samples, Fed-DNN-Debugger focuses on the problem of error weights caused by improper training processes.

*2.3. Metadata Capture.* Metadata generated by the model during training or testing are extremely important for model debugging. Auxiliary debugging tools [21–23] or data provenance studies [30–32] extract metadata using a data collection API during training. However, in the federated learning setting, manual extraction and analysis of training metadata are not allowed because of the data protection requirement. noWorkflow [33] is a nonintrusive method of metadata capture, which can collect metadata automatically, without inserting any data collection code into the machine learning script. However, noWorkflow collects massive amounts of metadata in deep learning scenarios. The method proposed in this paper is inspired by noWorkflow: it obtains metadata nonintrusively and combines it with the syntax of deep learning software to locate and extract target metadata during training or testing.

## 3. Background

*3.1. Federated Learning.* Suppose that there are $N$ data owners $\{F_1, F_2, \ldots F_N\}$, all of whom want to train machine learning models by merging their respective datasets $\{D_1, D_2, \ldots D_N\}$. The traditional approach is to pool these datasets together to train the model $M_{\text{tra}}$. Federated learning is a learning process in which data owners train the model $M_{\text{fed}}$ jointly under the coordination of a central server, such that any data owner $F_i$ does not disclose its dataset $D_i$ to the others. Defining the accuracy of models $M_{\text{tra}}$ and $M_{\text{fed}}$ as $\text{Acc}_{\text{tra}}$ and $\text{Acc}_{\text{fed}}$, respectively, federated learning aims to make $\text{Acc}_{\text{fed}}$ very close to $\text{Acc}_{\text{tra}}$ [34].

A typical assumption is that the $N$ participants are honest and the server is honest but curious. Therefore, no participant is allowed to leak information to the server [35]. The training process of federal learning usually includes the following four steps [34]:

**S1:** each participant trains the model locally and calculates the training gradient, uses encryption [35], differential privacy [36], or secret sharing [37] to mask the gradient, and sends the masked result to the server.

**S2:** the server accepts the gradient transmitted by each participant, to perform safe aggregation, without learning information about any participant.

**S3:** the server sends back the aggregated gradients to each participant.

**S4:** the participants accept the results returned by the server and update their local models with the decrypted gradients.

*3.2. Deep Neural Network Debugging.* Bugs in deep learning models can be divided into two categories [11]: *structure bugs* and *training bugs*. Structure bugs are generally caused by inappropriate model architecture, such as the number of hidden layers, the number of neurons per layer, and hyperparameters. Training bugs are usually caused by misconducted training processes, such as biased data, noisy data, and insufficient training. In this paper, we focus on training bugs. As discussed in Introduction, a deep learning model is essentially a set of neurons connected by a high-dimensional weight matrix, and the weights are updated during training. When there are problems such as biased data, noisy data, or inadequate training, some weights cannot be updated to the optimal state. Then, the error weights appear in the model, resulting in poor performance of the model.

The usual method of fixing training bugs is to modify the error weights in the model by retraining the model with more samples.

In practice, underfitting and overfitting are the most common problems of a deep learning model. Overfitting refers to a model learning the pattern and noise in the data to such an extent that it hurts the performance of the model on the test dataset. Underfitting is that the model neither learns from the training dataset nor generalizes well on the test dataset.

To measure the performance of a model, we define $\text{Tracc}(M_g^{\text{train}})$ as the training accuracy of label $g$ of model $M$ and define $\text{Teacc}(M_g^{\text{test}})$ as the test accuracy of label $g$ of model $M$. In practice, as shown in Figure 1(a), during the training process, if a model is overfitting, its training accuracy is higher than the test accuracy. If a model is underfitting, its training accuracy and test accuracy will be poor, as shown in Figure 1(b). Therefore, we say that a model has an overfitting or underfitting bug if it satisfies equations (1) or (2), respectively.

$$\exists g, \text{Tracc}\left(M_g^{\text{train}}\right) - \text{Teacc}\left(M_g^{\text{test}}\right) \geq \gamma, \tag{1}$$

$$\exists g, \text{Tracc}\left(M_g^{\text{train}}\right) \leq \theta, \text{Teacc}\left(M_g^{\text{test}}\right) \leq \theta, \tag{2}$$

where $\gamma$ and $\theta$ are predefined values based on concrete applications. If the test accuracy for both model and label becomes higher, and the accuracy of the label is no longer substantially lower than the model accuracy, we consider that the model is fixed.

## 4. Fed-DNN-Debugger Overview

Figure 2 presents the architecture of Fed-DNN-Debugger. The purpose of Fed-DNN-Debugger is to fix a trained federated model and improve its performance. The design of Fed-DNN-Debugger is based on a key insight: if the global model has error weights, some client models must have error weights. This is because all data are trained in client models and the server model only aggregates the results of each client model by FedAvg [1], without training. Thus, Fed-DNN-Debugger debugs the local model on each client. The error weights of the federated model can finally be repaired by fixing each client model. The detailed process is explained as follows:

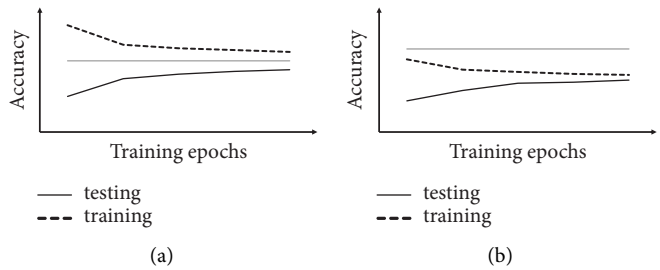


Figure 1: (a) Overfitting and (b) underfitting. The gray horizontal lines represent the desired accuracy.

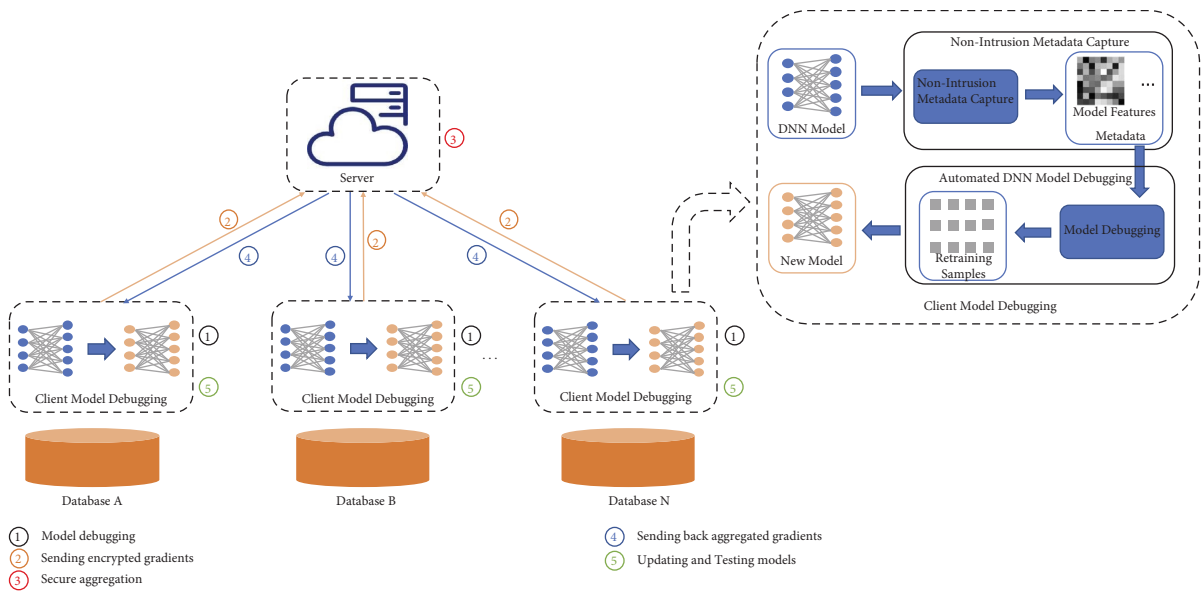


Figure 2: Architecture of Fed-DNN-Debugger. Debugging the client model to repair its weights. The gradients after local debugging are encrypted by homomorphic encryption and sent to the server. The server receives the encrypted gradients for secure aggregation and then returns the aggregated gradients to each client. The clients receive the returned gradients, update their local models, and perform tests.

(1) Each client locally performs the following debugging workflow in parallel. First, the client tests the model and automatically collects the metadata generated by the model, including the features generated by the model, prediction results, and model weights. The client then debugs the model according to the collected metadata and selects high-quality samples to fix the model. Finally, it computes gradients of the model during the retraining process.

(2) Each client masks the gradients with homomorphic encryption [35] and sends results to the server.

(3) The server accepts the masked gradients sent by each client and performs the secure aggregation algorithm [1].

(4) The server sends back the aggregated gradients to each client.

(5) Each client receives the encrypted gradients and updates its local model after decryption.

As shown in Figure 2, the debugging process of each client model is divided into two steps: NIMC and ANNMD. To collect the metadata automatically and transparently, NIMC constructs an execution dataflow by analyzing the abstract syntax tree (AST) of the deep learning script. It captures the metadata that helps debug from the dataflow, according to the syntax of the deep learning software (e.g., the syntax of PyTorch [38]), as described in Section 5. The purpose of ANNMD is to fix the client model according to the captured metadata. First, it selects a layer that guides repairing error weights and calculates the differential features [11] by the selected layer. Second, it scores candidate samples using the differential features and selects retraining data that have high scores. The details of ANNMD are explained in Section 6.

## 5. Nonintrusive Metadata Capture

The main purpose of NIMC is to collect the metadata that are useful for debugging, automatically and transparently. noWorkflow [33] is a commonly used method for automatic metadata collection from a Python script. It parses the AST of the entire Python script to obtain all function calls and variables and then collects their values during execution. noWorkflow is widely used in data provenance applications. Unfortunately, it is hard to apply this tool to deep learning

scenarios. The main reason is that noWorkflow collects metadata at a fine granularity and will collect massive amounts of metadata during model training or testing. However, most of the collected metadata are not relevant to model debugging. Collecting the metadata can significantly increase the complexity of debugging.

In this section, we propose NIMC, as shown in Figure 3, which collects metadata useful for DNN model debugging automatically during execution. In NIMC, there is no need to insert any metadata collection code into the deep learning script. Firstly, we design a code analysis module to obtain the deep learning dataflow from the python script using the AST tool. Then, we perform a metadata capture module to identify and collect the target metadata in the dataflow during the code execution. The implementations of each module in detail will be introduced as follows.

*5.1. Code Analysis.* The purpose of code analysis is to obtain the dataflow of the deep learning script. As shown in Figure 3, firstly, the script is compiled using the Python compiler to obtain the AST of the code. The AST is then analyzed to obtain the function definitions and their parameters. For example, the function call and method of an object in the Python script are represented as ast.Call and ast.Attribute, respectively, and the function definition and class definition correspond to ast.FunctionDef and ast.-ClassDef, respectively. The AST is searched to find all information related to function calls and variables. Finally, the dataflow of the code is composed of the searched function calls and variables.

*5.2. Metadata Capture.* Metadata capture is intended to collect the target metadata from the dataflow of the deep learning script. The dataflow constructed from the AST contains all the function calls and variables of the entire script. However, most of the metadata are useless for debugging. Therefore, the key to metadata capture is the method of identifying the target metadata from the dataflow.

We divide the training metadata into two categories. One type is generated by the deep learning software, such as the model weights and output features of the neural network. We call these the *software metadata*. Because deep learning software is implemented by different organizations and teams, their APIs and parameters are diverse. The other type is customized by the modeler, such as the testing or training accuracy and loss function. We call these the *custom metadata*.

Software metadata are defined by deep learning software, whose function names and parameters are fixed in each version. Therefore, this type of metadata can be identified in the dataflow by the syntax of the deep learning software. The custom metadata are defined by the modeler. Because the metadata are used when debugging the local model on the client side, the function name or variable name of the custom metadata should be marked by the modeler in advance.

To collect software metadata efficiently, we combine the syntax and APIs of the deep learning software to capture the metadata from the dataflow. In this paper, we take PyTorch [38] as an example. Figure 4(a) shows part of the code of a typical DNN model in PyTorch, and its dataflow is shown in Figure 4(b). In Figure 4(b), the blue nodes represent variables, and the orange nodes represent function calls. It is clear from the dataflow that the model-related function calls and variables show spatial locality. The nodes from Conv2d to loss are the target metadata for debugging; they constitute a path without branches in the dataflow. These characteristics guide collecting the target metadata. Because the API is defined by deep learning software, the function names are fixed in a certain version. Therefore, we traverse the dataflow according to these APIs to locate the target metadata. We have designed an algorithm for metadata identification, as shown in Algorithm 1. The input parameters of the algorithm include a dataflow graph of the code and a set of metadata names. The set of metadata names contains the API name from the deep learning software and the variable name specified by the modeler. The output of the algorithm is a list of target metadata. First, the depth-first search (DFS) algorithm [39] is executed to traverse the dataflow graph to obtain the list of nodes (line 1 of Algorithm 1). According to the characteristics of the dataflow discussed above, because a connected subgraph of the dataflow contains all target metadata, the DFS algorithm arranges all target metadata nodes adjacent to each other in list $G$; this is convenient for the subsequent identification of metadata. The target metadata list is then initialized (line 2 of Algorithm 1). Each node is checked; if either the node or the previous node is in the metadata name set, the node is added to the target metadata list (lines 3–7 of Algorithm 1). Finally, the target metadata list is returned. After identifying the target metadata from the dataflow, NIMC extracts them from the dataflow during code execution.

The advantages of NIMC are as follows. First, from the perspective of modeling, data scientists only need to focus on the machine learning task itself instead of metadata collection and model debugging, which will lower the threshold for modeling in federated learning and improve its efficiency. Second, from the perspective of privacy protection, nonintrusive metadata collection (NIMC) and automatic neural network model debugging (ANNMD) can avoid data scientists from manually collecting and analyzing privacy metadata, further reducing the risk of privacy leakage.

## 6. Automated Neural Network Model Debugging

In this section, we describe the ANNMD method, which repairs the error weights automatically and efficiently. Since the model is trained on the client and aggregated on the server, ANNMD automatically debugs the local model on each client. Given a model that has overfitting or underfitting problems for a label, ANNMD selects a hidden layer to generate features of correctly classified samples and misclassified samples for the label (Section 6.1). Based on the generated features, the differential features are then constructed for various types of problem (Section 6.2). Finally, the differential features are used to select retraining samples and repair the error weights (Section 6.3).

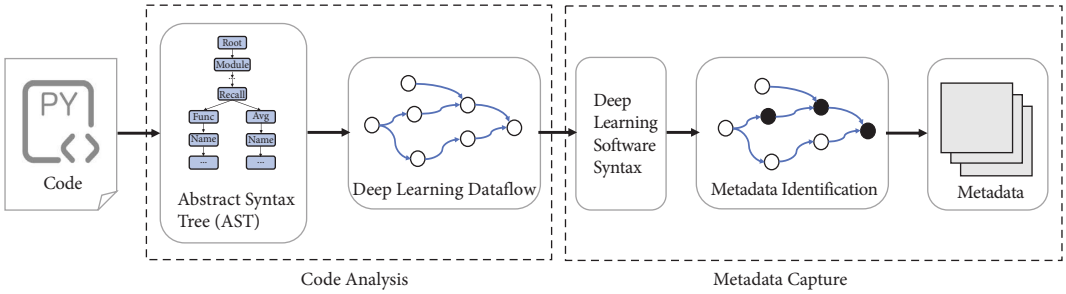


Figure 3: Workflow of nonintrusive metadata capture. The code analysis module is used to obtain the deep learning dataflow by compiling the Python script. The metadata capture module is performed to identify and collect the target metadata during code execution.

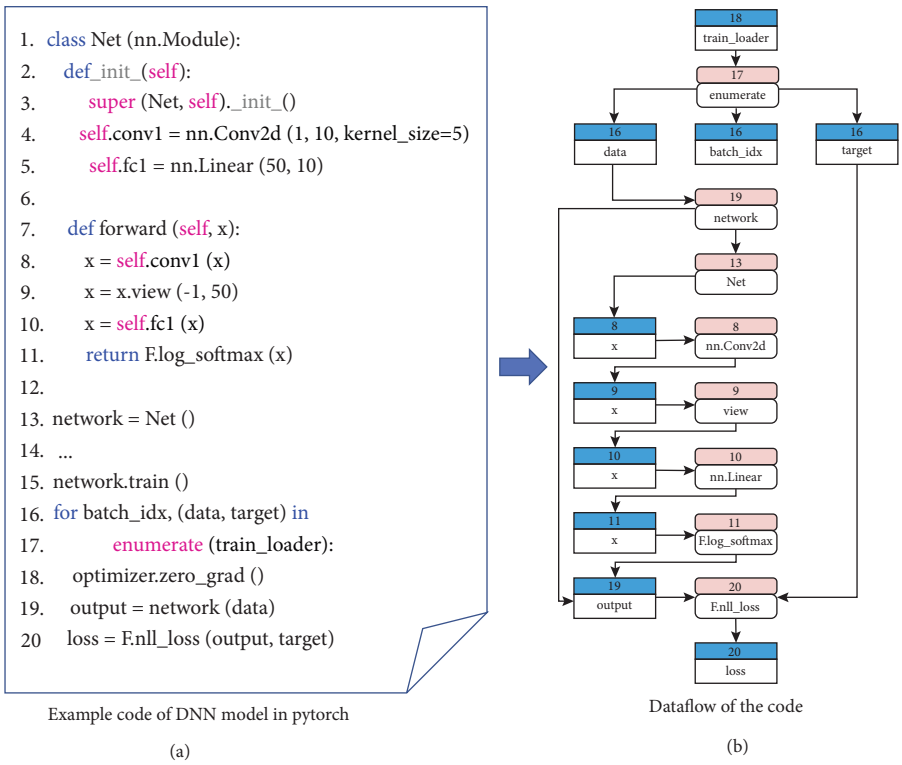


Figure 4: Example DNN code and dataflow in PyTorch. (a) Shows a segment of DNN code in PyTorch. (b) Shows the dataflow of the example code in (a). The blue nodes in the dataflow denote variables, and the orange nodes represent function calls in the script. The number of each node in the dataflow represents its line number in the code.

*6.1. Target Layer Selection.* We directly select the last hidden layer (the layer before the output layer) as the target layer. This layer is connected to the output layer, which is used for feature fusion and prediction. The output of the last hidden layer can be regarded as the final extracted features, which determine the performance of the model. Our experimental results show that the effect of selecting the last hidden layer as the target layer is similar to, or even better than, the effect of selecting the layer selected by MODE [11], and it consumes much less time.

*6.2. Differential Feature.* Differential features identify the features that guide problems; they can be achieved by comparing the correctly classified and misclassified features [11]. Differential features provide guidance for searching for samples that can fix the model. Given a label $l$ that we hope to debug, there are three groups of samples: $CI_l$ represents the correctly predicted samples for $l$, $MI_l$ denotes the samples that are misclassified as $l$, and $WI_l$ represents the samples of $l$ that are misclassified as other labels. For each group of samples, the features generated by the target layer are denoted by $FCI_l$ (for $CI_l$), $FMI_l$ (for $MI_l$), and $FWI_l$ (for $WI_l$).

*6.2.1. Underfitting.* To solve the underfitting problem for label $l$, we need to add more samples with features that are unique to label $l$ and eliminate the faulty features (which cause misclassification). To add more samples with features that are unique to label $l$, we calculate differential features $DFCI$ using correctly classified samples $CI$, as follows:

**Require:** dataflow graph $D_{\text{dataflow}}$, metadata name set $S_{\text{metadata}}$;
**Ensure:** target metadata list $T_{\text{metadata}}$
(1) $G \longleftarrow DFS(D_{\text{dataflow}})$
(2) $T_{\text{metadata}} = []$
(3) **for** $i = 1, \ldots, \text{len}(G)$ **do**
(4) **if** $G[i]$.name in $S_{\text{metadata}}$ **or** $G[i-1]$.name in $S_{\text{metadata}}$ **then**
(5) $T_{\text{metadata}}$.add$(G[i]$.name$)$
(6) **end if**
(7) **end for**
(8) Return $T_{\text{metadata}}$

Algorithm 1: Metadata identification algorithm.

$$DFCI_{l,k}[i] = FCI_l[i] - FCI_k[i], \tag{3}$$

where $DFCI_l[i] = DFCI_{l,k}[i]$, with $k \neq l, k \in L$, and abs $(DFCI_{l,k}[i])$ is minimized. $DFCI_{l,k}[i]$ denotes the difference of feature $i$ between labels $l$ and $k$.

To eliminate the faulty features for label $l$, we computes differential features $DFMI$, as follows:

$$DFMI_l[i] = FMI_l[i] - FCI_l[i], \tag{4}$$

where $DFMI_l[i]$ represents the difference of the feature $i$ between the faulty and correct classifications of label $l$.

*6.2.2. Overfitting.* Overfitting problems are mainly caused by biased training samples, so we need more samples with feature diversity to fix the model [11]. The misclassified samples guide the most-needed features, so we compute the differential features $DFWI_l$ using these samples.

$$DFWI_l[i] = MFWI_l[i] - FCI_l[i], \tag{5}$$

where $MFWI_l[i] = FWI_{l,k}[i]$, with $k \neq l$, $k \in L$, and abs $(FWI_{l,k}[i])$ is maximized. $FWI_{l,k}[i]$ represents the features for the samples of label $l$ that are misclassified as label $k$. $MFWI_l[i]$ denotes the maximal value that features $i$ for label $l$.

*6.3. Retraining Sample Selection.* After generating the differential features, we select retraining samples that can repair error weights. The algorithm for selecting retraining samples calculates a score to evaluate the similarity between the features of candidate samples and the differential features. The top $n$ samples, ranked by score, are then selected as the retraining samples.

Each sample is fed to the model to obtain a feature vector $sf$ from the selected target layer. For a given sample and a differential feature $\mathrm{d}f$, a score is calculated to measure the contribution of the model fixing. We combine a variety of measures of vector similarity, including Canberra distance, cosine distance, Euclidean distance, Manhattan distance, and Minkowski distance. These methods take into account the characteristics of feature space distribution. To combine the advantages of these methods when making decisions for selecting samples, the score is calculated as follows:

$$\text{score} = \frac{1}{N}\sum_{i}^{N} \text{score}_i, \tag{6}$$

$$\text{score}_i = \begin{cases} \text{normalize}(d_i(\text{sf}, \text{df})), & d_i \in LS \\ 1 - \text{normalize}(d_i(\text{sf}, \text{df})), & d_i \in LL, \end{cases} \tag{7}$$

where $d_i$ represents the $i$-th measure of vector similarity, sf represents the vector of the sample feature, and df represents the vector of the differential feature. normalize() represents the normalization method, which maps the value of the vector similarity to the range [0, 1].

$$\text{normalize}(d) = \frac{d - d_{\min}}{d_{\max} - d_{\min}}. \tag{8}$$

The score of each sample is calculated by equations (6)–(8), which combine the multiple similarity measures mentioned above. Equation (6) alone is not sufficient because we cannot simply add the values calculated by each method, for the following reasons. First, the range of values calculated by each method is different. For example, the range of values calculated by *cosine distance* is [0, 1], but the range of *Euclidean distance* is [0, +∞). If the values from each measure were added directly, the sample selection would be affected by the measure with the larger value. Second, for some measures, a larger value corresponds to a greater similarity between vectors. We have marked this type of method as **LS**, with others marked as **LL**. For example, a smaller Euclidean distance corresponds to greater similarity between the vectors, whereas the dot product has the opposite meaning. Therefore, we first convert all the values to the same form, whereby a larger value represents greater similarity between the vectors, and then normalize the result, as shown in equation (7).

After calculating the score of each sample, the sample with the highest score is selected as the high-quality sample to fix the model. It is worth noting that Fed-DNN-Debugger does not retrain using only samples selected by the score because that would introduce new problems [11]. Therefore, random samples are added for training, together with selected samples. The experiments reported in this paper show that the best ratio between selected and random samples is 3 : 7.

## 7. Debugging Algorithm

Algorithm 2 describes the debugging algorithm of Fed-DNN-Debugger. In practice, the debugging algorithm is mainly divided into two parts: server-orchestrated training (lines 1–12 in Algorithm 2) and client debugging (lines 14–22 in Algorithm 2). In each iteration process on the server side, we first test the model accuracy on the test set. If the test accuracy does not reach the target (line 3), the model is sent to each client for further debugging (lines 4–5); otherwise, the debugging ends (line 9). Each client accepts the model parameters from the server and updates its local model (line 15). Then, we input the local training dataset to the model and perform the NIMC method, described in Section 5, to obtain the metadata generated by the model (line 16). The last hidden layer's output features and prediction results are captured by the NIMC method. The retraining dataset is obtained by debugging the model using the ANNMD method (line 17), described in Section 6, and then the model is retrained using the stochastic gradient descent (SGD) algorithm (lines 18–21). Finally, the parameters of the debugged model are sent to the server for safe aggregation (line 22). The server accepts the model parameters transmitted by each client (lines 4–6), averages the model parameters of each client (line 7), and then goes to the next round of debugging.

*7.1. Privacy Analysis.* Our Fed-DNN-Debugger protects the local data of each client from being leaked during the debugging process. First, during the server-orchestrated training process, the training process strictly follows the standard federated learning training [1], as shown in Algorithm 2. Therefore, no local training data are transmitted other than the client model. Second, during the local debugging process, no local metadata will be collected and analyzed manually. To prevent information leakage caused by manual collection and analysis of private metadata, Fed-DNN-Debugger transparently and automatically collects metadata and debugs neural network models on each client side, without embedding any metadata collection and debugging code in training scripts. The entire debugging process is performed locally, and no metadata will be transmitted to the server. Thus, there is no information leakage during the local debugging process. It is worth noting that the research on preventing privacy leakage in the federated learning training process is a hot topic, but it is out of the scope of this paper.

*7.2. Communication and Computation Complexity.* Since Fed-DNN-Debugger introduces NIMC and ANNMD on the vanilla federated learning framework [1], additional computation has occurred. As shown in Algorithm 2, each client only shares the model and does not share any other intermediate results. Thus, our Fed-DNN-Debugger does not bring any additional communication overhead. In this section, we only focus on the additional computation introduced by Fed-DNN-Debugger, compared to vanilla federated learning.

On the server side, as shown in Algorithm 2, there is no additional computation. Like traditional federated learning, the server accepts and aggregates models from various clients. On the client side, as shown in Algorithm 2, the additional computation includes NIMC and ANNMD. For NIMC, the main computation is target metadata identification shown in Algorithm 1, which involves dataflow graph traversal and metadata matching. In Algorithm 1, the computation complexity of dataflow traversal (line 1 of Algorithm 1) is $O(N_G + N_E)$, where $N_G$ is the number of vertex of dataflow graph $G$ and $N_E$ is the number of edge of graph $G$. The computation complexity of metadata matching (lines 3–7 of Algorithm 1) is $O(N_S N_G)$, where $N_S$ represents the length of metadata name set $S_{\text{metadata}}$. Thus, the total computation complexity of NIMC is $O(N_G + N_E + N_S N_G)$. For ANNMD, the main computation includes differential feature generation and retraining sample selection. The computation complexity of differential features generation is $O(N_i)$, where $N_i$ is the number of samples in client $i$. The computation complexity of retraining sample selection is $O(N_i \log N_i)$. This is because sample selection involves sorting the data. Therefore, the total computation complexity of ANNMD is $O(N_i + N_i \log N_i)$. As a result, the total additional computation complexity of Fed-DNN-Debugger in each client is $O(N_G + N_E + N_S N_G + N_i + N_i \log N_i)$.

## 8. Experiments and Results

We implemented a prototype of Fed-DNN-Debugger on PySyft and PyTorch [38]. In the experiments, we aimed to answer the following research questions:

**RQ1:** how effective is Fed-DNN-Debugger in fixing a federated model?

**RQ2:** how efficient and effective is Fed-DNN-Debugger in automated debugging of neural network models?

**RQ3:** how efficient is Fed-DNN-Debugger in non-intrusive metadata capture?

### *8.1. Experimental Setup*

*8.1.1. Implementation and Deployment.* We set up our experiment in a distributed computing network equipped with GPUs. There were four nodes: one representing the server and the other three nodes representing clients (Client-1, Client-2, and Client-3), which could represent organizations in the real world. Each node was configured with an NVIDIA Tesla P100 GPU card. Our code implementation was based on PyTorch [38] 1.4.0, PySyft 0.2.5, and Python 3.7.6. For simplicity, we assumed that all the clients participated in the debugging process.

*8.1.2. Dataset*

*(1) MNIST.* For classification tasks, we used the MNIST [40] dataset of handwritten digits, which contains 60,000 training samples and 10,000 test samples. Each sample is a grayscale

**Require:** total number of users $N \in \mathbb{N}$, global training rounds $T_g \in \mathbb{N}$, local training rounds $T_l \in \mathbb{N}$, a buggy model $w$, training dataset $S_{\text{train}}$, test dataset $S_{\text{test}}$, target accuracy $\text{acc}_{\text{target}}$;
**Ensure:** well debugged model $w$;
**Server-orchestrated training:**//Server side
**for** each round $t$ from 0 to $T_g$ **do**
  $\text{acc}_{\text{test}} \longleftarrow$ (test model $w^t$ using $S_{\text{test}}$)
  **if** $\text{acc}_{\text{test}} \leq \text{acc}_{\text{target}}$ **then**
    **for** each user $n$ from 0 to $N$ in parallel **do**
      $w_n^{t+1} \longleftarrow$ ClientDebugger(n, $w^t$)
    **end for**
    $w^{t+1} \longleftarrow 1/N \sum_{n=0}^{N} w_n^{t+1}$
  **else**
    **return** $w^t$
  **end if**
**end for**
**return** $w^t$ **ClientDebugger** $(n, w^0)$://Client side
$w \longleftarrow w^0$
$f_{\text{last}}, p \longleftarrow$ Metada taCapture (Mode lCal $(w, S_{\text{train}})$)//**Nonintrusive Metadata Capture**
$S_{\text{train}}' \longleftarrow$ DLDebugger $(f_{\text{last}}, p, S_{\text{train}})$//**Automated Neural Network Model Debugging**
**for** each round $t$ from 0 to $T_l$ **do**
  $B \longleftarrow$ ($S_{\text{train}}'$ split into $k$ size $B$ batches)
  $w \longleftarrow SGD(B, w)$
**end for**
**return** $w$

Algorithm 2: Fed-DNN-Debugger.

image with a size of $28 \times 28$ pixels, representing one of the ten digits from 0 to 9.

*(2) CIFAR-10.* For object recognition tasks, we used the CIFAR-10 [41] dataset, which contains 60,000 color images with a size of $32 \times 32$ pixels. The dataset includes 50,000 training samples and 10,000 test samples, which are divided into ten categories.

To simulate the distributed data distribution in the real world, we partitioned the original training set over three clients in the non-IID (non-independent and identically distributed) setting, where Client-1 contains labels 0–3, Client-2 has labels 4–6, and Client-3 has labels 7–9. Each client split the training dataset into two parts: training dataset (80%) and validation dataset (20%). The training dataset was used for training, and the validation dataset was used for validating the model during training. Moreover, the original test dataset was used to evaluate the performance of the federated model.

*8.1.3. Models and Baseline.* We selected four different models to train on the same dataset. The four models had different neural network architectures. From Model-1 to Model-4, the number of layers increased, as shown in Table 1. Models 1–3 come from [42], and Model-4 is the famous VGG [43].

To answer RQ1, the baseline method was FedAvg [1] with the same number of randomly selected samples. To answer RQ2, we conducted a comparative experiment with MODE [11], for automated debugging of neural network models, and compared the performance and efficiency of the two methods when debugging a single model. To answer RQ3, we used noWorkflow [33] as the baseline method.

Table 1: Structure of DNN models.

| Model-1 | Model-2 | Model-3 | Model-4 |
|---|---|---|---|
| | | | Conv (64) × 2 |
| | | | MaxPooling |
| Conv (24) | Conv (32) × 2 | Conv (32) × 3 | Conv (128) × 2 |
| MaxPooling | MaxPooling | BatchNormalization | MaxPooling |
| Conv (48) | Conv (64) × 2 | Conv (64) × 3 | Conv (256) × 3 |
| MaxPooling | MaxPooling | BatchNormalization | MaxPooling |
| Flatten () | Flatten () | Flatten () | Conv (512) × 3 |
| Dense (256) | Dense (512) | Dense (128) | MaxPooling |
| Dense (10) | Dense (10) | Dense (10) | Flatten () |
| Softmax | Softmax | Softmax | Dense (4096) × 2 |
| | | | Dense (10) |
| | | | Softmax |

*8.2. Debugging Federated Models.* To answer RQ1, we prepared two sets of experiments for debugging federated models. Given a federated model with overfitting or underfitting problems, one group of experiments used Fed-DNN-Debugger to debug the model in the federated learning framework, and the other group executed the FedAvg method with the same number of randomly selected samples. The experimental results revealed that Fed-DNN-Debugger significantly improved the model performance and effectively fixed the federated model (Table 2).

Table 2: Debugging federated models.

| Dataset | Models | Problem type | Origin | | FedAvg [1] | | Fed-DNN-Debugger | | Improvement | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | MAcc (%) | LAcc (%) | MAcc (%) | LAcc (%) | MAcc (%) | LAcc (%) | MAcc (%) | LAcc (%) |
| MNIST | Model-1 | Overfitting | 91 | 81.31 | 94.53 | 94.12 | 96.1 | 97.95 | 1.57 | 3.83 |
| | | Underfitting | 93 | 82.48 | 94.38 | 92.87 | 98.3 | 98.98 | 3.92 | 6.11 |
| | Model-2 | Overfitting | 92.42 | 81.42 | 95.88 | 95.38 | 96.39 | 96.71 | 0.51 | 1.33 |
| | | Underfitting | 93.88 | 86.97 | 95.44 | 94.17 | 98.37 | 98.27 | 2.93 | 4.1 |
| | Model-3 | Overfitting | 88.65 | 73.5 | 94.59 | 92.99 | 96.4 | 97.23 | 1.81 | 4.24 |
| | | Underfitting | 91.54 | 86.35 | 95.85 | 94.05 | 97.77 | 98.88 | 1.92 | 4.83 |
| | Model-4 | Overfitting | 94.89 | 88.30 | 96.69 | 94.36 | 97.23 | 98.15 | 0.54 | 3.79 |
| | | Underfitting | 88.17 | 85.87 | 96.72 | 95.44 | 97.86 | 98.54 | 1.14 | 3.1 |
| CIFAR-10 | Model-1 | Overfitting | 66.51 | 49.7 | 64.81 | 45.4 | 68.61 | 64.48 | 3.8 | 19.08 |
| | | Underfitting | 68.07 | 48.8 | 69.58 | 56.4 | 72.75 | 70.4 | 3.17 | 14 |
| | Model-2 | Overfitting | 70.94 | 61 | 71.87 | 62.2 | 72.91 | 71.49 | 1.04 | 9.29 |
| | | Underfitting | 72.04 | 50.8 | 68.07 | 67.9 | 76.51 | 76.1 | 8.44 | 8.2 |
| | Model-3 | Overfitting | 80.57 | 68.67 | 82.85 | 82.89 | 84.86 | 84.68 | 2.01 | 1.79 |
| | | Underfitting | 75.66 | 65.9 | 75.66 | 75.2 | 82.56 | 79.9 | 6.9 | 4.7 |
| | Model-4 | Overfitting | 77.52 | 66.3 | 78.68 | 73.2 | 80.05 | 76.61 | 1.37 | 3.41 |
| | | Underfitting | 84.58 | 74.1 | 85.22 | 78 | 86.95 | 84.7 | 1.73 | 6.7 |
| Average improvement | | | | | | | | | 2.68 | 5.21 |

Table 2 shows, from left to right, the dataset, model, problem type, and test accuracy for both the model and the specific problem label (columns 1–5). Columns 6–9 of Table 2 show the test accuracy of the model and the problem label using Fed-DNN-Debugger and FedAvg, respectively. The last two columns show the improvement in the accuracy of the model and the problem label.

From Table 2, we make some observations. First, compared with the method of randomly increasing training samples, Fed-DNN-Debugger was more effective in fixing the federated model. After fixing the model, the test accuracy of the model and the problem label were improved, indicating that Fed-DNN-Debugger did not reduce the model performance. Second, we found that randomly increasing samples could slightly improve the accuracy of the model, but the accuracy of some labels was not improved, which shows that it could not fix the model or it introduced new problems. Fed-DNN-Debugger did not introduce new problems while fixing the model. As we can observe, the accuracy of the four models with Fed-DNN-Debugger exceeded that of FedAvg by 0.51%–8.44%, with an average of 2.68%, and for the problem label by 1.33%–19.08%, with an average of 5.21%. In all cases, the accuracy of the federated model was improved. FedAvg performed poorly in some cases because the retraining samples were not selected for repairing the error weights. Fed-DNN-Debugger, with the bug fixing setting, outperformed FedAvg.

Figure 5 shows the debugging effect of Fed-DNN-Debugger on each client and the server. Figures 5(a)–5(d) show the debugging process and the performance of Model-1 to Model-4 on the MNIST dataset. The $x$-axes of the subgraphs represent the debugging round, and the $y$-axes represent the test accuracy of the model. In the figure, the blue, orange, green, and red curves represent the test results on the server, Client-1, Client-2, and Client-3, respectively. The model on the server represents the federated model. As shown in Figure 5, the performance of the federated model improved with the improvements in the client models, and it took only a few rounds to converge. This shows that our idea of fixing the federated model by fixing the client models is feasible.

Two critical parameters of Fed-DNN-Debugger are the number of retraining samples and the proportion of selected high-quality samples in the retraining dataset. To obtain the two parameters, Table 3 shows the experimental results of the model performance with different proportions of the selected data in the retraining dataset. From left to right, the table shows the dataset, proportion of selected high-quality samples in the retraining dataset, and (in columns 3–11) the number of retraining samples. We can observe that the model achieved the best performance when the ratio was 0.3 and the number of retraining samples was 4000.

*8.3. Debugging a Single Model.* To answer RQ2, we conducted two sets of experiments for single-model debugging. Given a model with the overfitting or underfitting problem, one set of experiments used Fed-DNN-Debugger, and the other set of experiments used MODE [11]. We recorded the execution time and the accuracy of the model and the label in each group of debugging experiments. The experimental results show that Fed-DNN-Debugger significantly improved both the speed and the performance in single-model debugging.

Table 4 shows the experimental results of debugging a single model. From left to right, it shows the dataset, model, problem type, and the test accuracy for both the model and the specific label (columns 1–5). Columns 6–11 show the debugging time and the highest test accuracy of the model and the label, for both MODE and Fed-DNN-Debugger. Column 12 shows the execution time of MODE divided by that of Fed-DNN-Debugger. Columns 13 and 14 show the difference between the test accuracy (of the model and the label) of Fed-DNN-Debugger and that of MODE.

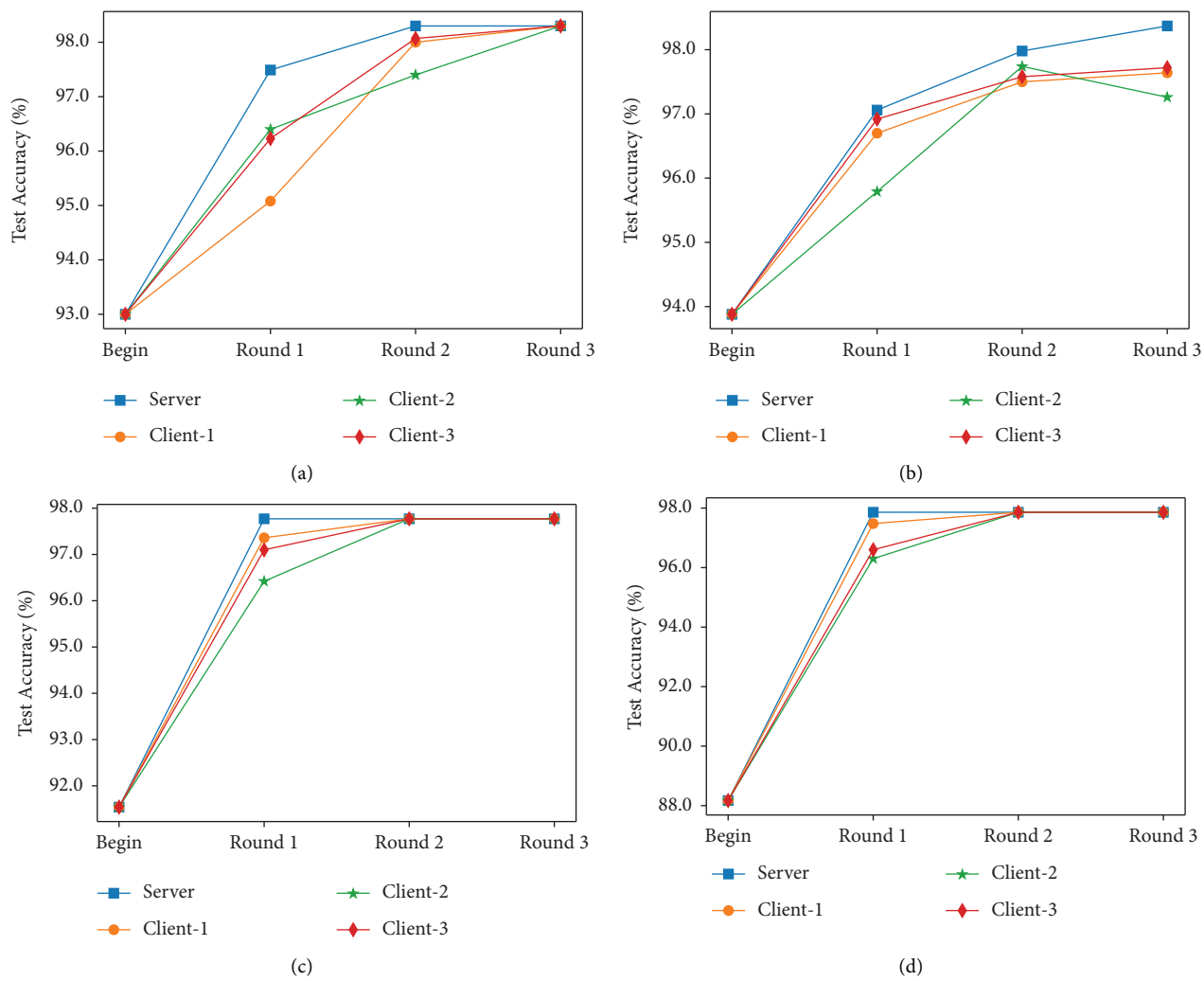


Figure 5: Effects of server and clients on the MNIST dataset. The $x$-axes of the subgraphs represent the debugging round, and the $y$-axes represent the test accuracy of the model. The test accuracies of the models on the server, Client-1, Client-2, and Client-3 are plotted in blue, orange, green, and red, respectively. (a) Debugging round of Model-1. (b) Debugging round of Model-2. (c) Debugging round of Model-3. (d) Debugging round of Model-4.

As shown in Table 4, Fed-DNN-Debugger was 23 times faster than MODE, on average. The main reason is that MODE needs to repeatedly fine-tune and test the model when locating the target layer that provides guidance for fixing the model. Fed-DNN-Debugger selects the last hidden layer as the target layer directly, saving substantial time. For debugging accuracy, Fed-DNN-Debugger performs better than MODE because MODE uses only the vector dot product to score samples for retraining. In contrast, Fed-DNN-Debugger integrates multiple measures of vector similarity, which evaluate the training samples from various perspectives. Therefore, the samples selected by Fed-DNN-Debugger are of higher quality.

*8.4. Nonintrusive Metadata Capture.* To answer RQ3, we executed Fed-DNN-Debugger and noWorkflow [33] to collect metadata during the test process. We recorded the size of the collected metadata and the time spent on metadata capture. The experimental results show that Fed-DNN-Debugger significantly improved the collection speed and reduced the storage space consumption.

Table 5 shows the size of the collected metadata on the MNIST and CIFAR-10 datasets. Clearly, the size of the metadata obtained by Fed-DNN-Debugger was far less than that obtained by noWorkflow. For instance, the metadata generated by Model-4 on the CIFAR-10 dataset collected by noWorkflow occupied 203 GB. However, the metadata captured by Fed-DNN-Debugger occupied just 1.499 GB. This is because Fed-DNN-Debugger captures only the metadata that are helpful for debugging, whereas noWorkflow obtains the return values of all function calls and variables generated by the entire script. These results show that, in the process of debugging a deep learning model, only a small number of metadata are useful for model debugging, whereas most of the metadata are useless.

Table 6 shows the collection time of the different models on the MNIST and CIFAR-10 datasets. Fed-DNN-Debugger is clearly more efficient than noWorkflow in extracting

Table 3: Effects of the proportion of selected high-quality samples in the retraining data and the number of retraining samples.

| Dataset | Ratio | 500 (%) | 1000 (%) | 1500 (%) | 2000 (%) | 2500 (%) | 3000 (%) | 3500 (%) | 4000 (%) | 4500 (%) |
|---|---|---|---|---|---|---|---|---|---|---|
| MNIST | 0.1 | 94.76 | 95.56 | 92.5 | 95.44 | 95.37 | 96 | 96.15 | 96.75 | 95.48 |
| | 0.15 | 93.85 | 90.89 | 94.35 | 92.27 | 96.32 | 95.32 | 95.25 | 96.42 | 94.94 |
| | 0.2 | 92.82 | 95.21 | 94.21 | 95.6 | 93.39 | 95.99 | 96.29 | 96.1 | 96.36 |
| | 0.25 | 94.65 | 94.03 | 94.58 | 95.35 | 96.27 | 95.9 | 94.95 | 96.09 | 95.56 |
| | 0.3 | 94.81 | 94.84 | 95.28 | 94.76 | 94.69 | 96.09 | 96.53 | **96.96** | 96.69 |
| | 0.35 | 93.36 | 92.13 | 95.03 | 94.35 | 96.08 | 96.35 | 95.84 | 96.39 | 94.81 |
| | 0.4 | 93.88 | 83.24 | 94.89 | 94.14 | 94.23 | 94.5 | 94.98 | 95.5 | 96.31 |
| | 0.45 | 93.89 | 93.67 | 93.46 | 94.44 | 93.61 | 95.13 | 93.96 | 96.41 | 96.64 |
| CIFAR-10 | 0.1 | 69.63 | 71.79 | 72.69 | 72.28 | 74.05 | 71.60 | 74.44 | 72.99 | 74.52 |
| | 0.15 | 71.53 | 69.98 | 73.43 | 73.27 | 73.68 | 72.69 | 67.20 | 73.04 | 74.64 |
| | 0.2 | 72.36 | 70.46 | 73.25 | 71.53 | 68.30 | 73.52 | 72.63 | 74.02 | 74.34 |
| | 0.25 | 71.88 | 71.70 | 72.52 | 72.83 | 72.01 | 74.09 | 72.04 | 74.07 | 73.95 |
| | 0.3 | 73.40 | 70.14 | 72.39 | 70.72 | 68.84 | 69.68 | 72.79 | **74.86** | 74.01 |
| | 0.35 | 70.91 | 69.24 | 70.91 | 70.82 | 62.61 | 74.08 | 70.47 | 74.03 | 74.12 |
| | 0.4 | 70.27 | 70.69 | 71.45 | 72.88 | 74.30 | 73.55 | 73.66 | 74.38 | 73.17 |
| | 0.45 | 71.31 | 66.23 | 70.91 | 65.67 | 72.83 | 73.32 | 74.08 | 73.74 | 72.70 |

The two bold values indicate the combination of parameters for the best model performance on these two datasets.

Table 4: Debugging a single model.

| Dataset | Models | Problem type | Origin | | | MODE | | | Fed-DNN-Debugger | | | Improvement | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | MAcc (%) | LAcc (%) | Time (s) | MAcc (%) | LAcc (%) | Time (s) | MAcc (%) | LAcc (%) | Faster | MAcc (%) | LAcc (%) |
| MNIST | Model-1 | Overfitting | 91.11 | 81.3 | 169.11 | 96.65 | 94.6 | 11.81 | 97.96 | 98.7 | 14.32 | 1.31 | 4.1% |
| | | Underfitting | 93.00 | 82.5 | 118.86 | 96.82 | 97.6 | 8.18 | 97.91 | 99.2 | 14.53 | 1.09 | 1.6% |
| | Model-2 | Overfitting | 92.42 | 81.4 | 144.96 | 92.42 | 81.4 | 15.2 | 98.16 | 97.7 | 9.53 | 5.74 | 16.3% |
| | | Underfitting | 93.88 | 87.0 | 194.45 | 96.96 | 96.6 | 10.84 | 97.81 | 97.0 | 17.94 | 0.85 | 0.4% |
| | Model-3 | Overfitting | 88.65 | 73.5 | 490.62 | 96.63 | 93.2 | 31.25 | 97.61 | 97.3 | 15.70 | 0.98 | 4.1% |
| | | Underfitting | 91.54 | 83.9 | 566.26 | 97.63 | 98.9 | 30.65 | 98.62 | 99.0 | 18.47 | 0.99 | 0.1% |
| | Model-4 | Overfitting | 94.89 | 88.3 | 863.60 | 96.97 | 95.3 | 191.03 | 98.14 | 96.2 | 4.52 | 1.17 | 0.87% |
| | | Underfitting | 88.17 | 83.9 | 1523.38 | 95.99 | 98.2 | 543.27 | 96.73 | 98.9 | 2.80 | 0.74 | 0.75% |
| CIFAR-10 | Model-1 | Overfitting | 66.51 | 49.7 | 251.67 | 67.73 | 51.1 | 7.1 | 69.72 | 61.9 | 35.44 | 1.99 | 10.8% |
| | | Underfitting | 68.07 | 48.8 | 187.31 | 69.30 | 41.5 | 6.9 | 72.40 | 68.6 | 27.14 | 3.10 | 27.1% |
| | Model-2 | Overfitting | 76.94 | 57.0 | 459.84 | 77.11 | 76.6 | 8.12 | 79.21 | 76.8 | 56.63 | 2.10 | 0.2% |
| | | Underfitting | 72.04 | 56.5 | 330.79 | 74.28 | 62.0 | 7.96 | 76.88 | 66.3 | 41.55 | 2.60 | 4.33% |
| | Model-3 | Overfitting | 80.57 | 65.4 | 743.02 | 82.66 | 78.3 | 13.92 | 88.23 | 86.7 | 53.38 | 5.57 | 8.4% |
| | | Underfitting | 75.66 | 60.1 | 584.86 | 82.14 | 75.3 | 14.22 | 87.89 | 86.2 | 41.13 | 5.75 | 10.87% |
| | Model-4 | Overfitting | 77.52 | 65.5 | 494.04 | 85.04 | 85.9 | 148.77 | 88.16 | 87.9 | 3.32 | 3.12 | 2.08% |
| | | Underfitting | 84.58 | 63.7 | 1359.60 | 87.26 | 87.2 | 88.96 | 89.00 | 87.3 | 15.28 | 1.74 | 0.11% |
| Average improvement | | | | | | | | | | | 23.23 | 2.42 | 5.76% |

Table 5: Size of collected metadata.

| Dataset | Model | noWorkflow [33] (GB) | Fed-DNN-Debugger (GB) |
|---|---|---|---|
| MNIST | Model-1 | 9.6 | 0.135 |
| | Model-2 | 25 | 0.188 |
| | Model-3 | 37 | 0.176 |
| | Model-4 | 147 | 0.736 |
| CIFAR-10 | Model-1 | 13 | 0.235 |
| | Model-2 | 33 | 0.3215 |
| | Model-3 | 51 | 0.3089 |
| | Model-4 | 203 | 1.499 |

metadata. For example, the time required by noWorkflow in Model-4 on the CIFAR-10 dataset was 15771.6 s, whereas Fed-DNN-Debugger took just 113 s.

To summarize, the size of the metadata collected by Fed-DNN-Debugger was only 0.6%–1.8% of that collected by noWorkflow, with an average of 0.9%. Moreover, the

Table 6: Execution time of metadata capture.

| Dataset | Model | noWorkflow [33] (s) | Fed-DNN-Debugger (s) |
|---|---|---|---|
| MNIST | Model-1 | 326.19 | 30.221 |
| | Model-2 | 805.41 | 56.339 |
| | Model-3 | 1204.61 | 90.311 |
| | Model-4 | 4806.39 | 111.145 |
| CIFAR-10 | Model-1 | 1322.6 | 30.385 |
| | Model-2 | 2560.2 | 58.67 |
| | Model-3 | 4242.9 | 96.5281 |
| | Model-4 | 15771.6 | 113.25 |

execution time of noWorkflow was 10–139 times greater than that of Fed-DNN-Debugger, with an average of 42 times. These results show that Fed-DNN-Debugger is more efficient than noWorkflow in the deep learning debugging scenario.

## 9. Conclusion

In this paper, we have presented Fed-DNN-Debugger, the first debugging system that can automatically and efficiently fix DNN models in federated learning. Fed-DNN-Debugger fixes the overall model by fixing each client model because only client models are trained on data. Fed-DNN-Debugger consists of two modules for debugging a client model: NIMC and ANNMD. NIMC analyzes the data flows of client models with deep learning software syntax to efficiently collect metadata that are helpful for debugging. It does not insert any code for metadata collection into modeling scripts. ANNMD automatically scores samples according to metadata and searches for high-quality samples. Models are retrained with the selected samples to fix training bugs. Experimental results on the CIFAR-10 and MNIST datasets and four DNN models have shown that Fed-DNN-Debugger can improve the test accuracy by 8%.

## Data Availability

The data used to support the findings of this study are included within the article.

## Conflicts of Interest

The authors declare that they have no conflicts of interest.

## Acknowledgments

This study was supported in part by the National Natural Science Foundation of China (grant no. 61872110), Guangdong Provincial Key Laboratory of Novel Security Intelligence Technologies (no. 2022B1212010005), and Major Key Project of PCL (grant nos. PCL2022A03, PCL2021A02, and PCL2021A09).

## References

[1] B. McMahan, E. Moore, D. Ramage, S. Hampson, and y A. Blaise Aguera, "Communication-efficient learning of deep networks from decentralized data," in *Proceedings of the Artificial Intelligence and Statistics*, pp. 1273–1282, PMLR, New York, NY, USA, April 2017.

[2] T. Li, A. K. Sahu, A. Talwalkar, and V. Smith, "Federated learning: challenges, methods, and future directions," *IEEE Signal Processing Magazine*, vol. 37, no. 3, pp. 50–60, 2020.

[3] X. Yin, Y. Zhu, and J. Hu, "A comprehensive survey of privacy-preserving federated learning: a taxonomy, review, and future directions," *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–36, 2021.

[4] A. Vaid, S. K. Jaladanki, J. Xu et al., "Federated learning of electronic health records to improve mortality prediction in hospitalized patients with covid-19: machine learning approach," *JMIR medical informatics*, vol. 9, no. 1, 2021.

[5] T. K. Dang, X. Lan, J. Weng, and M. Feng, "Federated learning for electronic health records," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 13, no. 5, pp. 1–17, 2022.

[6] Yi Liu, J. J. Q. Yu, J. Kang, D. Niyato, and S. Zhang, "Privacy-preserving traffic flow prediction: a federated learning approach," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7751–7763, 2020.

[7] J. Pei, K. Zhong, M. A. Jan, and J. Li, "Personalized federated learning framework for network traffic anomaly detection," *Computer Networks*, vol. 209, Article ID 108906, 2022.

[8] X. Bai, H. Wang, L. Ma et al., "Advancing covid-19 diagnosis with privacy-preserving collaboration in artificial intelligence," *Nature Machine Intelligence*, vol. 3, no. 12, pp. 1081–1089, 2021.

[9] M. Jiang, Z. Wang, and Q. Dou, "Harmofl: harmonizing local and global drifts in federated learning on heterogeneous medical images," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 1, pp. 1087–1095, 2022.

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

[11] 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*, pp. 175–186, Lake Buena Vista, FL, USA, October 2018.

[12] H. Zhang and W. K. Chan, "Apricot: a weight-adaptation approach to fixing deep learning models," in *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 376–387, IEEE, San Diego, CA, USA, November 2019.

[13] S. Duan, C. Liu, Z. Cao, X. Jin, and P. Han, "Fed-dr-filter: using global data representation to reduce the impact of noisy labels on the performance of federated learning," *Future Generation Computer Systems*, vol. 137, pp. 336–348, 2022.

[14] Y. Sun, P. Yuan, and Y. Sun, "Mm-gan: 3d mri data augmentation for medical image segmentation via generative adversarial networks," in *Proceedings of the 2020 IEEE International Conference on Knowledge Graph (ICKG)*, pp. 227–234, IEEE, Nanjing, China, August 2020.

[15] D. Bhattacharya, S. Banerjee, S. Bhattacharya, B. U. Shankar, and S. Mitra, "Gan-based novel approach for data augmentation with improved disease classification," in *Advancement of Machine Intelligence in Interactive Medical Image Analysis*, pp. 229–239, Springer, Berlin, Germany, 2020.

[16] M. Ernst Tschuchnig, C. Ferner, and S. Wegenkittl, "Sequential iot data augmentation using generative adversarial networks," in *Proceedings of the ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal*

*Processing (ICASSP)*, pp. 4212–4216, IEEE, Barcelona, Spain, May 2020.

[17] D. Jia, W. Dong, R. Socher, L. J. Li, K. Li, and L. Fei-Fei, "Imagenet: a large-scale hierarchical image database," in *Proceedings of the 2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, IEEE, Miami, FL, USA, June 2009.

[18] B. Zhou, A. Lapedriza, A. Khosla, A. Oliva, and A. Torralba, "Places: a 10 million image database for scene recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 6, pp. 1452–1464, 2018.

[19] G. Tao, S. Ma, Y. Liu, Q. Xu, and X. Zhang, "Trader: trace divergence analysis and embedding regulation for debugging recurrent neural networks," in *Proceedings of the 2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 986–998, IEEE, Seoul, South Korea, June 2020.

[20] D. J. L. Lee, M. Stephen, D. Xin, A. Lee, S. Huang, and A. G. Parameswaran, "A human-in-the-loop perspective on automl: milestones and the road ahead," *IEEE Data Engineering Bulletin*, vol. 42, no. 2, pp. 59–70, 2019.

[21] P. Han, C. Wang, C. Liu, S. Duan, H. Pan, and P. Luo, "Securemldebugger: a privacy-preserving machine learning debugging tool," in *Proceedings of the 2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*, pp. 127–134, IEEE, Hong Kong, China, July 2020.

[22] N. Rauschmayr, V. Kumar, R. Huilgol et al., "Amazon sagemaker debugger: a system for real-time insights into machine learning model training," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 770–782, 2021.

[23] H. B. Braiek and F. Khomh, "Testing feedforward neural networks training programs," *ACM Transactions on Software Engineering and Methodology*, 2022.

[24] M. Y. Liu, X. Huang, J. Yu, T.-C. Wang, and A. Mallya, "Generative adversarial networks for image and video synthesis: algorithms and applications," *Proceedings of the IEEE*, vol. 109, no. 5, pp. 839–862, 2021.

[25] Z. Cai, Z. Xiong, H. Xu, P. Wang, W. Li, and Y. Pan, "Generative adversarial networks: a survey toward private and secure applications," *ACM Computing Surveys*, vol. 54, no. 6, pp. 1–38, 2021.

[26] I. Goodfellow, J. Pouget-Abadie, M. Mirza et al., *Generative Adversarial Nets. In Advances in Neural Information Processing Systems*, Curran Associates, Inc, Red Hook, NY, USA, 2014.

[27] A. Li, L. Zhang, J. Wang, F. Han, and X.-Y. Li, "Privacy-preserving efficient federated-learning model debugging," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2291–2303, 2022.

[28] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," in *Proceedings of the International Conference on Machine Learning*, pp. 1885–1894, PMLR, New York, NY, USA, July 2017.

[29] A. Schioppa, P. Zablotskaia, D. Vilar, and A. Sokolov, "Scaling up influence functions," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, pp. 8179–8186, Palo Alto, CA, US, July 2022.

[30] S. Schelter, J.-H. Boese, J. Kirschnick, T. Klein, and S. Seufert, "Automatically tracking metadata and provenance of machine learning experiments," in *Proceedings of the Machine Learning Systems Workshop at NIPS*, Long Beach, CA, USA, January 2017.

[31] J. F. Pimentel, J. Freire, L. Murta, and V. Braganholo, "A survey on collecting, managing, and analyzing provenance from scripts," *ACM Computing Surveys*, vol. 52, no. 3, pp. 1–38, 2019.

[32] M. Hossein Namaki, A. Floratou, F. Psallidas et al., "Vamsa: automated provenance tracking in data science scripts," in *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1542–1551, Virtual Event, CA, USA, August 2020.

[33] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "noworkflow: a tool for collecting, analyzing, and managing provenance from python scripts," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1841–1844, 2017.

[34] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: concept and applications," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 10, no. 2, pp. 1–19, 2019.

[35] Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 5, pp. 1333–1345, 2017.

[36] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pp. 1310–1321, Denver, CO, USA, October 2015.

[37] B. Keith, V. Ivanov, B. Kreuter et al., "Practical secure aggregation for privacy-preserving machine learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1175–1191, Dallas, TX, USA, October 2017.

[38] N. Ketkar, "Introduction to pytorch," in *Deep Learning with python*, pp. 195–208, Springer, Berlin, Germany, 2017.

[39] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[40] Li Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141-142, 2012.

[41] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Handbook of Systemic Autoimmune Diseases*, vol. 1, no. 4, 2009.

[42] GoogleCloudPlatform, "Mnist tutorial," 2018, https://github.com/GoogleCloudPlatform/tensorflow-without-a-phd/tree/master/tensorflow-mnist-tutorial.

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Proceedings of the 3rd International Conference on Learning Representations, ICLR 2015*, ICLR, San Diego, CA, USA, May 2015.