Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 6887040, 15 pages
https://doi.org/10.1155/2022/6887040

Research Article

Local Epochs Inefficiency Caused by Device Heterogeneity in Federated Learning

Yan Zeng,[1,2,3] **Xin Wang**,[4] **Junfeng Yuan**,[1] **Jilin Zhang**,[1,2,3] **and Jian Wan**[1]

[1]*School of Computer Science and Technology, Hangzhou Dianzi University, Hangzhou 310018, China*
[2]*Key Laboratory of Complex Systems Modeling and Simulation Ministry of Education, Hangzhou 310018, China*
[3]*Zhejiang Engineering Research Center of Data Security Governance, Hangzhou 310018, China*
[4]*HDU-ITMO Joint Institute, Hangzhou Dianzi University, Hangzhou 310018, China*

Correspondence should be addressed to Jilin Zhang; jilin.zhang@hdu.edu.cn

Received 5 August 2021; Revised 17 November 2021; Accepted 29 November 2021; Published 6 January 2022

Academic Editor: Jinbo Xiong

Federated learning is a new framework of machine learning, it trains models locally on multiple clients and then uploads local models to the server for model aggregation iteratively until the model converges. In most cases, the local epochs of all clients are set to the same value in federated learning. In practice, the clients are usually heterogeneous, which leads to the inconsistent training speed of clients. The faster clients will remain idle for a long time to wait for the slower clients, which prolongs the model training time. As the time cost of clients' local training can reflect the clients' training speed, and it can be used to guide the dynamic setting of local epochs, we propose a method based on deep learning to predict the training time of models on heterogeneous clients. First, a neural network is designed to extract the influence of different model features on training time. Second, we propose a dimensionality reduction rule to extract the key features which have a great impact on training time based on the influence of model features. Finally, we use the key features extracted by the dimensionality reduction rule to train the time prediction model. Our experiments show that, compared with the current prediction method, our method reduces 30% of model features and 25% of training data for the convolutional layer, 20% of model features and 20% of training data for the dense layer, while maintaining the same level of prediction error.

1. Introduction

In recent years, with the rapid development of 5G, "Interconnection of All Things" has become a trend of information technology. The speed of information circulation on the Internet has reached an unprecedented level. The rapid information circulation has brought a dramatic increase in data and promoted the development of big data and artificial intelligence technology. However, the interconnection of more devices brings higher security risks. How to enjoy the benefits brought by artificial intelligence on the premise of ensuring data privacy has become a challenge.

In 2016, McMahan et al. [1] first proposed the concept of federated learning to respond to the challenge. Multiple clients can jointly train the model under the coordination of a central server or service provider in federated learning. Each client uses its data to train the local model and uploads the parameters of local model to the server for model aggregation to achieve the global model. Therefore, the original data of each client is stored locally without exchange or transmission. At present, federated learning has been widely applied to optimize the user experience on the premise of protecting privacy in Google, Apple, and other enterprises. For example, Google has widely used federated learning in Gboard [2], Pixel mobile phone [3], and Android Message [4], so as IOS13 [5] of Apple.

In the FedAvg algorithm proposed by McMahan et al., the central server generates a global model after aggregation in each synchronization and then distributes the global model to some clients which are selected randomly. When the clients received the global model, they use their own data to train their local models with the parameters of global model in the specified epochs. After the local models are trained, the clients will send them to the server, and the

server will execute model aggregation with the weighted average strategy. The global model will gradually converge after several synchronizations. According to the FedAvg, the update of global model in each synchronization can be expressed by formula (1), where $x_g^{(t)}$ represents the parameters of the global model in the t-th synchronization of model aggregation, n_i is the amount of client i's data, and n denotes the amount of all clients' data. η is the client learning rate, τ_i represents the number of client i's local updates, which can be obtained by $\lfloor E \bullet n_i/B \rfloor$, E is the number of local epochs, B is the mini-batch size of the client, $x_i^{(t,k)}$ represents the parameters of local model in k-th local update on i-th client after the $t-$th synchronization, and g_i denotes the local model's parameters gradient of client i.

$$x_g^{(t+1)} - x_g^{(t)} = -\sum_{i=1}^{m} \frac{n_i}{n} \bullet \eta \sum_{k=0}^{\tau_i - 1} g_i\left(x_i^{(t,k)}\right), \text{where } \tau_i = \left\lfloor \frac{E \bullet n_i}{B} \right\rfloor. \tag{1}$$

In each synchronization of global model updating, FedAvg sets the same number of local epochs E for all clients participating in model aggregation, which will lead to inefficient training as the difference in training speed. A lot of other federated learning algorithms are also set in the same way, such as FedProx [6]. Although FedNova [7] has implemented the training of the global model under the situation of different local epochs, it randomly sets the number of local epochs, which may set a small number of local epochs for clients with high training speed, and leads to the idling problem. But, FedNova gives us an inspiration: if we can predict the model training time of clients, the local epochs will be dynamically set according to the predicted training time, which can reflect the training speed of clients.

As the idling of faster clients in federated learning will prolong the training of global model, we propose to predict the training time of deep learning models on heterogeneous clients to guide dynamically set the number of local epochs. In the deep learning task, the training time may be affected by the amount of training data and the setting of hyperparameters. We call the factors in training data and hyperparameters that may affect the training time as model features. Justus et al. [8] have trained a multilayer perceptron (MLP) to predict the training time of layers in the neural network using the model features and training time collected on different GPUs. They divide model features into layer features and predict the training time of the complete model by accumulating the prediction results of multiple layers. However, when the structure of the model is very complex or the number of layers is very large, collecting many features in each layer will increase the burden of the system and hinder the convergence progress of global model. What is more, when a new device is added to the federated learning system, it is necessary to collect a large amount of high dimension training data on this device to tune the current prediction model, which is very time-consuming and will lead to long-term failure of training time prediction for the new device.

To solve these problems, we propose a method based on deep learning to reduce the number of model features and the amount of training data required by training time prediction. We first design a neural network to extract the influence of model features on training time, which can accurately interpret the relationship between model features and training time. Then, we propose a dimensionality reduction rule to extract the key features based on the influence of model features on training time. A large number of experiments show that compared with the current prediction method, the features of the convolutional layer and dense layer can be reduced by 30% and 20%, respectively, and the error of training time prediction still maintains the original level. At the same time, 25% convolutional layer training data and 20% dense layer training data are reduced, which speeds up the adaptation of the time prediction model to the new device.

The rest of this paper is arranged as follows: in Section 2, we discuss the related work in time prediction; in Section 3, we introduce the method proposed in this work, including our neural network and dimensionality reduction rule. We also provide an algorithm to dynamically set the number of local epochs in this section. To verify our work, we set up a large number of experiments and interpret the experimental results in Section 4. Finally, we provide the summary of all work in Section 5.

2. Related Work

At first, machine learning regression algorithms are often used to predict time series, such as linear regression, random forest, and GBDT. Edelman et al. [9] use a linear regression model to predict the execution time of surgery; Wang et al. [10] train regression decision trees to predict the arrival time of buses by using the nearest neighbor-based random forest algorithm; Cheng et al. [11] use GBDT to predict traffic time in different time ranges and find the variables which have a great impact on prediction error. These regression methods have good universality, and they can be used in many fields. But, the error range of their prediction is very large, so they can only be applied to scenarios with low sensitivity to time fluctuations.

To narrow the error range of time series prediction, some scholars have proposed the method with specific domain knowledge to predict time series. It constructs mathematical models to achieve time series prediction, by studying the calculation characteristics of the specific domain such as PALEO [12] and Optimus [13] method. PALEO is a method to predict the computing time by counting the number of floating-point operations. It counts the number of floating-point operations required in a model training epoch and multiplies the number by a scale factor to predict the training time. However, PALEO assumes that the whole training process of the model is linearly related to the number of floating-point operations, ignoring some operations that are not, such as parameter transmission. Unlike PALEO, Optimus mathematically summarizes the factors that affect model training, establishes a performance model to evaluate the training speed, and can predict the model

convergence according to online resources. Compared with the regression methods, these works reduce the prediction error range of model training time to a certain extent, but the mathematical model established for the training is fuzzy and it ignores some factors which contribute greatly to the training time, resulting in instability of the prediction.

Because of the excellent performance of deep learning models, researchers began to use deep learning methods for predicting time series, and trying to further reduce the error of time series prediction. Xu et al. [14] creatively combine linear regression and deep belief network (DBN) to predict time series; PreVIous [15] trains the MLP model to predict the inference time of convolutional neural networks according to the throughput and energy consumption of the Internet of Things vision device; Petersen et al. [16] design a neural network mixed with convolutional layers and LSTM layers to accurately predict the bus arrival time. These works have achieved high prediction accuracy, but their application in the training time prediction of deep learning models is limited by the specific model structure. Their time prediction models can only predict those network structures contained in their training data (such as VGG [17], ResNet [18], or user-defined network). When a network with a new structure is encountered, their models need to be retrained, that is, they cannot apply to other new deep learning models. Although Fathom [19] has proved that the inference time of a model can be estimated by another model with a similar structure and known performance, its prediction is very rough, and it is still to be proved that whether this method can be used in the prediction of training time. In order to accurately predict the training time of networks with different structures, Justus et al. divide the neural network into layers and classify these layers (such as convolutional layer and dense layer) according to the structural characteristics, then collect the layer features and train an MLP model to predict the training time of a single layer in the neural network, which can achieve high prediction accuracy. This method has good generality. When a network with a new structure is encountered, it only needs to predict the training time of layers according to the layer model features, and then the training time of the whole model can be predicted by accumulating the training time of layers. However, there are some problems when Daniel Justus' method is applied to federated learning: (1) the relationship between model features and training time is not accurate. They assumed that almost every model feature is necessary for training time prediction, including features that have no or little impact on the final result. (2) Too many unimportant features need to be collected. When the neural network is very deep, collecting features in every layer will increase the burden of the federated learning system, and it is usually difficult to obtain all details of clients' models. (3) High training cost caused by too much redundant training data. Unimportant features produce a lot of redundant training data, which increases the transfer training cost of the time prediction model on the newly added device and reduces the training efficiency of the global model.

To solve the above problems, we propose a training time prediction method based on deep learning, which can reduce the required model features and training data on the premise of ensuring low prediction error and improve the feasibility of practical application in federated learning. The contributions of this paper are as follows:

(1) We design a neural network to extract the influence of model features on training time according to the characteristics of the deep learning model, which provides an effective analysis of the relationship between model features and training time

(2) We propose a dimensionality reduction rule to extract the key features that have a great impact on training time according to the influence of features, which can reduce the number of features required for predicting model training time without loss of prediction accuracy. By using the dimensionality reduction rule, 7 dimensions are extracted from convolutional layer features (10 dimensions in total), and 4 dimensions are extracted from dense layer features (5 dimensions in total)

(3) We train the time prediction model using dimension-reduced datasets. Compared with the method of Justus et al., the training data of the convolutional layer is reduced by 25%, and the training data of the dense layer is reduced by 20%, with the error of prediction remaining at the same level

3. Methodology

In this section, we will introduce the overall process and technical details of extracting the influence of model features on training time, dimensionality reduction, and the algorithm of dynamically setting the number of local epochs. First, we prove the feasibility of accumulating the layers' training time for the prediction of the whole network by interpreting the calculation process of training. And we describe the layer features based on the work of Justus et al. in detail. Second, we introduce the structure of the neural network (Here we name our neural network weights model), which is designed for extracting the influence of model features on the training time. Third, we propose the dimensionality reduction rule to extract the key features which have a great impact on training time, based on the influence of model features. Finally, we provide a representative algorithm for dynamically setting the number of local epochs.

3.1. Feature Analysis. One training of neural network consists of forward propagation and backward propagation. With the widespread use of Batch Normalization [20] which can speed up the convergence of neural networks, it usually has to perform a batch of forward propagation before one backward propagation. A complete round of training (including multiple batches) of a neural network in the training set is called an epoch. Generally, a model with high accuracy needs to be trained many epochs until the model converges. At present, the method of setting the number of epochs is based on the experience of deep learning engineers.

Table 1: The description of layer features.

Kind of features	Features	Description
Common features	Activation function	The activation function of neuron output, the common ones are sigmoid, tanh, and relu, etc.
	Optimizer	The optimization method of the model, the common ones are SGD, Adadelta, Adagrad, momentum, Adam, and RMS prop, etc.
Dense features	Number of inputs	Since the MLP layers are fully connected, the input of each layer comes from the output of the previous layer.
	Number of neurons	The number of neurons.
Convolutional features	Matrix size	The size of the input data.
	Kernel size	The size of convolutional kernel.
	Input depth	The number of input channels.
	Output depth	The number of output channels.
	Stride size	The convolution step size of convolution kernel.
	Input padding	The number of edge padding after convolution.
Hardware features	GPU clock speed	GPU clock cycle speed.
	GPU memory bandwidth	GPU bandwidth.
	GPU core count	The number of GPU processing units, which represents the number of CUDA cores in NVIDIA GPU.

It needs to set the different number of epochs for different models to achieve the specified accuracy. Therefore, it becomes a challenge to accurately predict the training time of models with the different number of epochs. In addition, since the structures of models are heterogeneous, the training time will be significantly different. For example, the training of the convolutional layer is usually more time-consuming than the dense layer. Therefore, it is also a challenge to accurately predict the training time for models with different structures. To solve these problems, Justus et al. proposed a method to predict the training time of different layers in a batch. The training time of the whole model in one batch can be obtained by accumulating the prediction results of layers. And the total training time can be predicted by accumulating the training time in batches. Ulteriorly, we prove the feasibility of predicting the whole model's training time through layers combined with the training characteristics and structural characteristics of the deep learning model.

Usually, a neural network needs to be trained repeatedly on the training set many times, which has obvious iteration characteristics. According to the iteration, the time required for model training can be expressed as formula (2), where E represents the number of epochs, M represents the number of batches in an epoch, and T_b denotes the training time in a batch. The total number of batches in E epochs can be obtained by $\lfloor E \bullet n/B \rfloor$, n is the amount of training data, and B is the size of a batch (i.e., batch size).

$$T = E \bullet M \bullet T_b = \left\lfloor \frac{E \bullet n}{B} \right\rfloor \bullet T_b. \tag{2}$$

One training of the neural network consists of a batch of forward propagation and one backward propagation. Therefore, the time cost of the propagation can be expressed as formula (3), t_{forward} represents the time cost of forward propagation, t_{backward} represents the time cost of backward propagation, and x_i is the i-th training data in a batch.

$$T_b = \left(\sum_{i=1}^{B} t_{\text{forward}}(x_i) \right) + t_{\text{backward}}. \tag{3}$$

Combining formula (2) and formula (3), the training time of a deep learning model can be described as the following formula:

$$T = \left\lfloor E \bullet \frac{n}{B} \right\rfloor \bullet \left[\left(\sum_{i=1}^{B} t_{\text{forward}}(x_i) \right) + t_{\text{backward}} \right]. \tag{4}$$

A neural network is composed of many layers, and the output of the current layer is used as the input of the next layer. Besides the iteration characteristic of the training process, the neural network also has obvious hierarchical structure characteristics. According to this hierarchy characteristic, a complete neural network can be divided into layers, and its forward and backward propagation can be expressed as formulas (5) and (6) on layer level. t^l_{forward} and t^l_{backward}, respectively, represent the time cost of layer l 's forward and backward propagation, and m denotes the number of model layers.

$$t_{\text{forward}}(x_i) = \sum_{l=1}^{m} t^l_{\text{forward}}(x_i), \tag{5}$$

$$t_{\text{backward}} = \sum_{l=1}^{m} t^l_{\text{backward}}. \tag{6}$$

Combining formula (4) with formulas (5) and (6), the training time formula of the complete model with the layer's

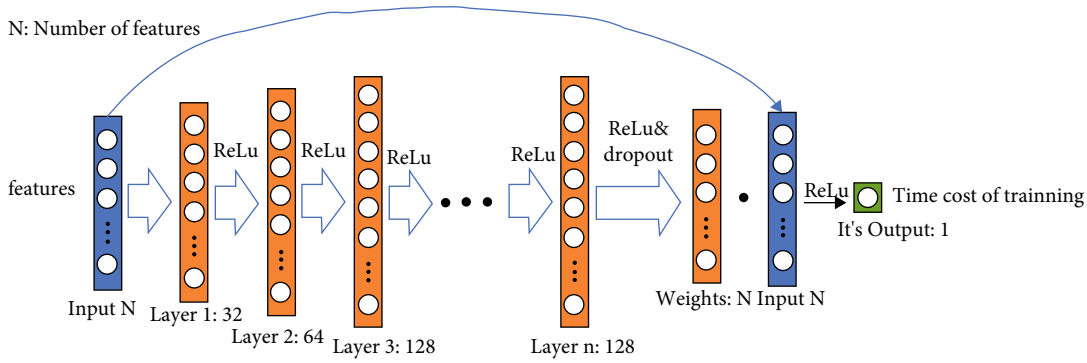

Figure 1: The structure of weights model.

Table 2: Hyperparameter settings.

Hyperparameter	Value
Initial learning rate	0.1
The decay period of learning rate	40
The decay ratio of learning rate	$2^{-\lfloor epoch/40 \rfloor}$
L2 regularization	0.00001
Dropout	0.2
Batchsize	128
Epoch	300
Activation	ReLu
Optimizer	Adam
Loss	Mean squared error

training time as the unit can be derived, as shown in the following formula.

$$T = \left\lfloor E \bullet \frac{n}{B} \right\rfloor \bullet \left[\left(\sum_{i=1}^{B} \sum_{l=1}^{m} t^{l}_{\text{forward}}(x_i) \right) + \sum_{l=1}^{m} t^{l}_{\text{backward}} \right]. \quad (7)$$

In summary, the training time of a single layer can be used as the basic unit of the whole model's training time. Therefore, for models with number of different epochs, the total training time of the model can be obtained by accumulating training time in a batch; for models with different structures, one batch training of the model can be obtained by accumulating the forward and backward propagation time of layers, well solved the two challenges in training time prediction of models.

In order to predict the training time of a single layer, it is necessary to analyze the layer features of neural networks. First of all, we classify the layer features into common features, dense features, convolutional features, and hardware features according to the device characteristics and computing characteristics. And then we extract the features according to the categories. The layer features are shown in Table 1. Since the deep learning model contains a large number of convolutional layers and dense layers, we mainly focus on the convolutional layer and dense layer in this paper. And we trained the time prediction model for the convolutional layer and the dense layer, respectively, based on the data of layer features and training time, which are collected on six different types of GPUs (P100, V100, K40, K80, M60, and 1080ti).

3.2. Model Design. In the previous section, we analyzed the layer features which may affect the training time of neural networks by parsing the structure of different layers. However, we find that collecting layer features will increase the system overhead for deep-seated neural networks. For example, in terms of the ReNet101 with 100 convolutional layers and 1 dense layer, if we collect 10 features of the convolutional layer and 5 features of the dense layer, we finally need to collect 1005 features to predict the training time of ResNet101. For clients with low computing power, the process of predicting will take a lot of time.

What is more, since there are a large-scale of heterogeneous devices in federated learning, we cannot use the time prediction model to predict the training time of models for newly added devices whose types are not in the set of preset device types. To predict the training time for the new device, we need to tune the parameters of the time prediction model based on the training data collected on this device. But it needs a long time to collect a large number of high dimension training data for tuning the time prediction model. The prediction model cannot quickly adapt to the new device, and the number of local epochs is set to be a fixed value for a long time, which may lead to the clients remaining idle with high training speed.

For cutting down the time cost of predicting and accelerating the adaptation of the time prediction model to new devices, it is necessary to reduce the dimension of layer features and redundant training data. In the case of ensuring high prediction accuracy, we choose to exclude the features that have no or little impact on training time. So, the influence of model features on training time needs to be analyzed.

In order to extract the influence of features, we abstract the relationship between model features and training time as $f(x) = wx$, where x represents the model features, $f(x)$ denotes the training time of a single layer, and w denotes the weights of features which can be treated as the influence of features. It should be noted that due to different value ranges of features, and w cannot represent the real influence

Input: W weights model, N the number of dataset, F feature set, X_k dataset k, M_d the amount of dataset d
Output: Θ the set of key features
Initialize Θ
For each dataset
    $w_{i,j} \longleftarrow$ GetWeights (W, X_k)
    $r_{i,j} \longleftarrow$ rank $(w_{i,j})$
End for
For j in F **do**
    $meanrank_j \longleftarrow (1/N)\sum_{d=1}^{N}(1/M_d)\sum_{i=1}^{M_d} r_{i,j}$
    $meanstd_j \longleftarrow (1/N)\sum_{d=1}^{N} RankStd(d, j)$
    If $meanrank_j > s$ and $meanstd_j < r$ **then**
        AddNewFeature (Θ, j)
    End if
End for
Return Θ

Algorithm 1: Dimensionality reduction.

Input: The K clients are indexed by i; B local minibatch size, T Coomunication time window, M time prediction model.
Server executes:
initialize $x_g^{(0)}$
for each round t, t =1,2,…,N **do**
    $m \longleftarrow$ Max $(C \bullet K, 1)$
    $S_t \longleftarrow$ (random set of m clients)
    for each client $i \in S_t$ in parallel: **do**
        $\eta d_i^{(t)} \longleftarrow$ **ClientUpdate**$(i, x_g^{(t)}, M)$
    end for
    $x_g^{(t+1)} \longleftarrow x_g^{(t)} - \tau_{eff}^{(t)} \sum_{i=1}^{m} (n_i/n) \eta d_i^{(t)}$
end for
ClientUpdate$(i, x_g^{(t)}, M)$:
$x_i^{(t)} \longleftarrow x_g^{(t)}$ // *Client receives the global model*
$N_i \longleftarrow$ (number of batches per epoch divided by B)
$f \longleftarrow$ GetFeatures$(i, x_i^{(t)})$
$T_{train} \longleftarrow$ GetTrainingTime(M, f)
$E_i \longleftarrow \lfloor T/T_{train} \rfloor$
for each epoch from e to E_i **do**
    for each batch from k to N_i **do**
        $x_i^{(t+1,k)} \longleftarrow x_i^{(t,k)} - \eta g_i^{(t,k)}$
        $d_i^{(t,k)} \longleftarrow d_i^{(t,k)} + (1/E_i * N_i) g_i(x_i^{(t,k)})$
        $d_i^{(t)} \longleftarrow d_i^{(t)} + d_i^{(t,k)}$
    end for
end for
return $\eta d_i^{(t)}$ to the Server

Algorithm 2: Dynamically set number of local epochs.

of features when simply taking the original feature data as x. The value of x should be the standardized feature data.

In related work, we introduced some machine learning regression models, including linear models and nonlinear models. The linear regression model can directly extract the features' weights w, but it underperforms in time prediction. The nonlinear models are difficult to extract w since their weights are implicitly dispersed in the model parameters. To get the weights of features accurately, we design a weights model which can use a neural network to explicitly extract features. The structure of the weights model is shown in Figure 1. See Table 2 for the hyperparameter settings of weights model.

As can be seen from Figure 1, the input of the weights model is the standardized feature data, and the output is the predicted training time. Each neuron of layers is

Table 3: The description of datasets.

Datasets	Description
P100_Conv	Convolutional layer feature dataset of P100.
P100_Dense	Dense layer feature dataset of P100.
V100_Conv	Convolutional layer feature dataset of V100.
V100_Dense	Dense layer feature dataset of V100.
K40_Conv	Convolutional layer feature dataset of K40.
K40_Dense	Dense layer feature dataset of K40.
All_Conv	Convolutional layer feature datasets of six different types of GPUs, including P100, V100, K40, K80, M60, and 1080ti (stacked dataset).
All_Dense	Dense layer feature datasets of six different types of GPUs, including P100, V100, K40, K80, M60, and 1080ti (stacked dataset).

activated by ReLu and then output. For ensuring the convergence of the model, the settings of layer 1 to layer N are consistent with the hidden layers of Justus et al.'s time prediction model. In order to learn the weights of features, we multiply the output of the weights layer and the input layer, and then output after ReLu activation. The weights of features w calculated from layer 1 to weights layer is multiplied by the feature data x to form $f(x) = wx$.

Different from the simple linear model whose w is fixed, the weight extracted by weights model will change with the input data, which can fit the training data better. In weights model, for each input data x_i, the output is $f(x_i) = g(x_i)x_i$, where $g(x_i)$ is a weight function that the weight will change with the input data. It can be obviously found that the weight $g(x_i)$ is obtained by the deep learning model, which means that it can not only benefit from the high accuracy of the nonlinear model but also use the output of weight layer to obtain the weight data explicitly. Justus et al. have proved that their MLP model has higher prediction accuracy than the linear regression model, but it cannot characterize the performance of our weights model, which is the reconstructed MLP. Therefore, we conduct the comparative experiments to prove the superiority of weights model (see Section 4.2 for the results).

3.3. Dimensionality Reduction Rule. We have introduced the weights model $g(x_i)$ which used to extract the weights of features in the last section. As the weights $g(x_i)$ will change with the input data x_i, the order of features' influence (i.e., weights ranking) may fluctuate. For example, for input data x_1, the feature batchsize has the greatest influence, but for data x_2, its influence may be the smallest. The prediction error will be further expanded and the influence of features cannot be measured uniformly because of the fluctuation of (x_i). Therefore, we use the average ranking of feature weights and the average standard deviation of weights ranking to comprehensively analyze the influence of features. The average ranking of feature weights can represent the overall contribution of features to training time and the average standard deviation of weights ranking can measure the fluctuation of weights. According to these two metrics, we propose a dimensionality reduction rule to extract the

Table 4: Layer features in the convolutional layer.

Features	Description
Batchsize	The size of a batch.
Elements_matrix	The number of elements of input.
Elements_kernel	The number of elements of convolutional kernel.
Channels_in	The number of input channels.
Channels_out	The number of output channels.
Padding	The number of edge padding.
Strides	The step size of convolution.
Use_bias	Whether to use bias, 0: not use 1: use.
Opt_SGD	Whether to use SGD optimizer, 0: not use 1: use.
Opt_Adadelta	Whether to use Adadelta optimizer, 0: not use 1: use.
Opt_Adagrad	Whether to use Adagrad optimizer, 0: not use 1: use.
Opt_Momentum	Whether to use momentum optimizer, 0: not use 1: use.
Opt_Adam	Whether to use Adam optimizer, 0: not use 1: use.
Opt_RMSProp	Whether to use RMSProp optimizer, 0: not use 1: use.
Act_relu	Whether to use ReLu activation, 0: not use 1: use.
Act_tanh	Whether to use tanh activation, 0: not use 1: use.
Act_sigmoid	Whether to use sigmoid activation, 0: not use 1: use.

Table 5: Layer features in the dense layer.

Features	Description
Batchsize	The size of a batch.
Dim_input	The dimension of input.
Dim_output	The dimension of output.
Opt_SGD	Whether to use SGD optimizer, 0: not use 1: use.
Opt_Adadelta	Whether to use Adadelta optimizer, 0: not use 1: use.
Opt_Adagrad	Whether to use Adagrad optimizer, 0: not use 1: use.
Opt_Momentum	Whether to use momentum optimizer, 0: not use 1: use.
Opt_Adam	Whether to use Adam optimizer, 0: not use 1: use.
Opt_RMSProp	Whether to use RMSProp optimizer, 0: not use 1: use.
Act_relu	Whether to use ReLu activation, 0: not use 1: use.
Act_tanh	Whether to use tanh activation, 0: not use 1: use.
Act_sigmoid	Whether to use sigmoid activation, 0: not use 1: use.

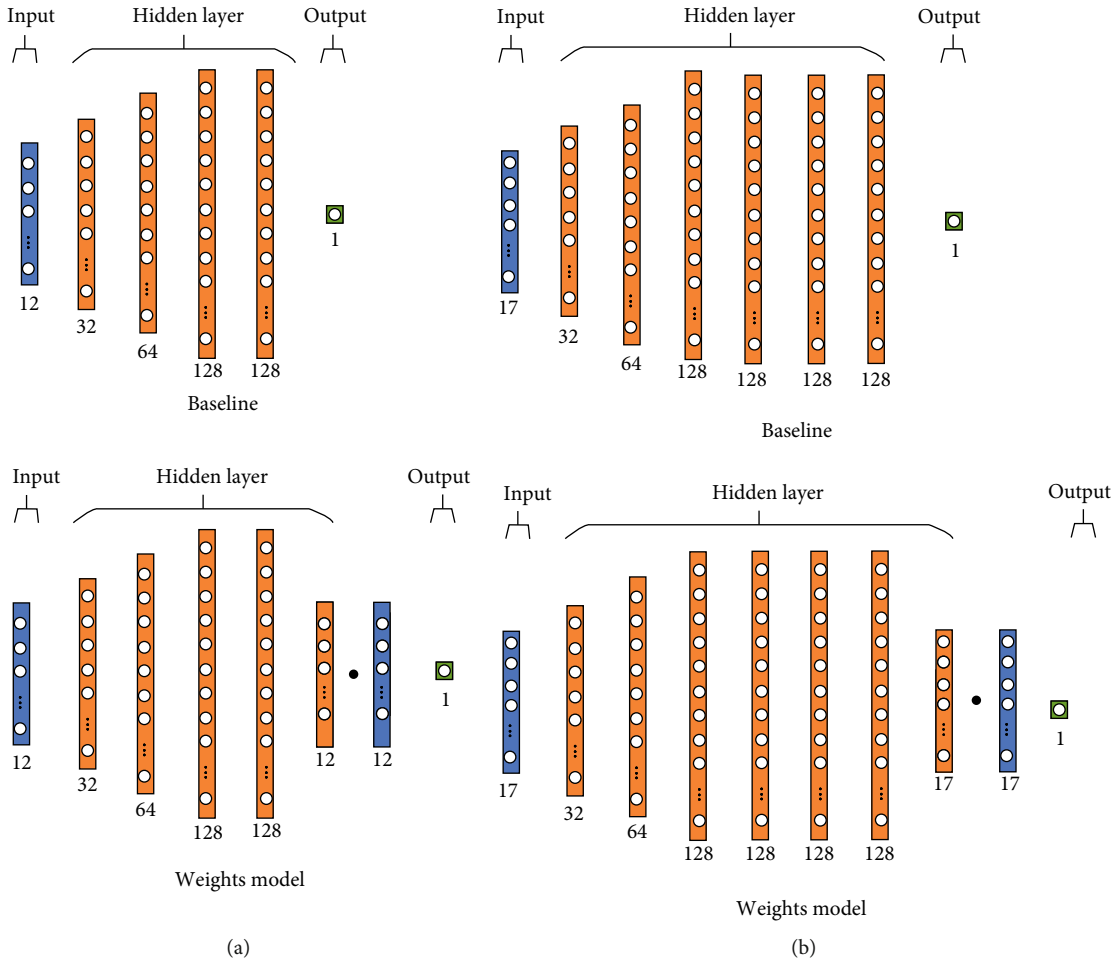

FIGURE 2: (a) The structures of baseline and weights model for single-GPU datasets. (b) The structures of baseline and weights model for stacked datasets.

key features that have a great overall influence on training time. Our analysis method and dimensionality reduction rule will be introduced in detail as follows.

Before analyzing the influence of feature weights, we first need to extract the weights of features by weights model based on multiple datasets described in Section 3.1. Then, we use formula (8) to calculate the ranking of the j-th feature's weight in weights data i, where $w_{i,k}$ represents the weight of the k-th feature in the weight data i, and n represents the total number of features.

$$\mathrm{Rank}(i,j) = 1 + \sum_{k=1}^{n} \max\left(\frac{\left||w_{i,k}| - |w_{i,j}|\right|}{|w_{i,k}| - |w_{i,j}|}, 0\right). \tag{8}$$

The standard deviation is an indicator which can reflect the extent of data dispersion. For measuring the fluctuation of the ranking of feature weights, we calculate the standard deviation of weights' ranking, and use $\mathrm{RankStd}(d,j)$ to represent the standard deviation of the j-th feature weight's ranking of dataset d.

Because of the device heterogeneity in federated learning, it is not universal to extract key features based only on the dataset generated by a single device. Therefore, we analyze the weights of features in many datasets collected from different GPUs. We use formula (9) to calculate the average ranking of feature weights to represent the overall contribution of features on heterogeneous devices and use formula (10) to obtain the average standard deviation of weights ranking for measuring the overall fluctuation extent of the weights ranking. Where N represents the number of datasets, and M_d represents the data volume of dataset d.

$$\mathrm{MeanRank}(j) = \frac{1}{N}\sum_{d=1}^{N}\frac{1}{M_d}\sum_{i=1}^{M_d}\mathrm{Rank}(i,j), \tag{9}$$

$$\mathrm{MeanRankStd}(j) = \frac{1}{N}\sum_{d=1}^{N}\mathrm{RankStd}(d,j). \tag{10}$$

MeanRank and MeanRankStd reflect the overall influence of features on training time from the perspective of contribution and fluctuation. In order to reduce the feature dimension, we formulate a unified dimensionality reduction rule according to MeanRank and MeanRankStd for extracting the key features which have a great overall impact on training time. The dimensionality reduction

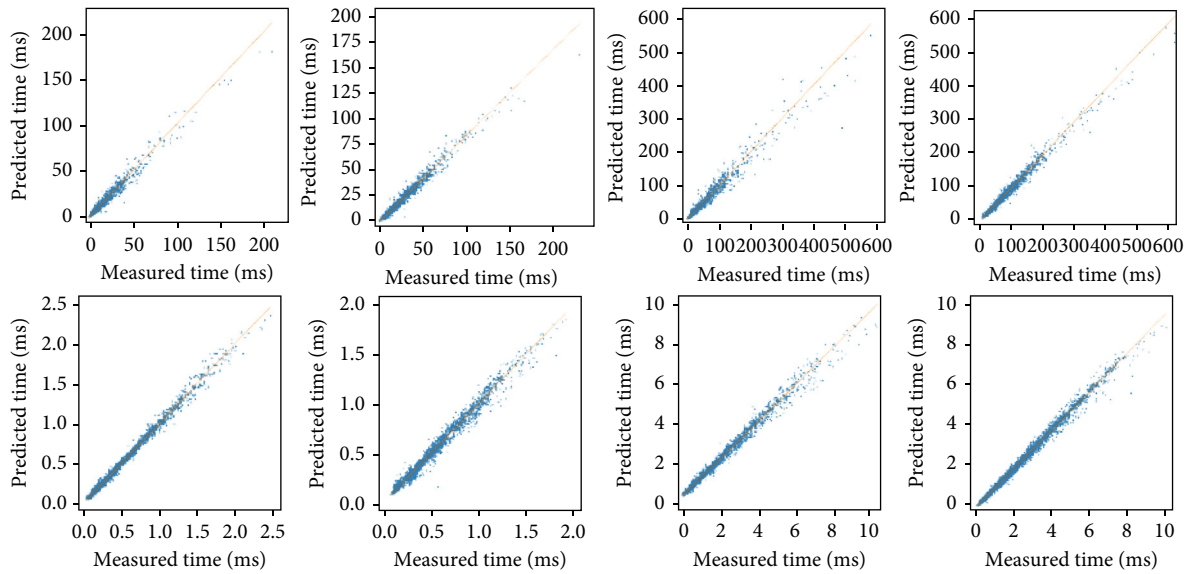

Figure 3: Predicted time vs. observed time. The upper of the figure: predicted time vs. observed time on the convolutional feature datasets, from left to right, is the results on *P100_Conv*, *V100_Conv*, *K40_Conv*, and *All_Conv*; the lower of the figure: predicted time vs. observed time on the dense feature datasets, from left to right, is the results on *P100_Dense*, *V100_Dense*, *K40_Dense*, and *All_Dense*.

Table 6: Weights model vs. linear regression model on *P100*, *V100*, and *K40*.

Datasets	RMSE	
	Weights model	Linear regression model
P100_Conv	3.09 ms	12.59 ms
V100_Conv	1.85 ms	9.46 ms
K40_Conv	11.67 ms	47.41 ms

rule can be expressed as formula (11), Θ represents the set of key features, F is the collection of all features, and s and r are constants, whose values need to be set according to the MeanRank and MeanRankStd. Note that the values of s and r are set empirically. After many experiments, we found that the effect of dimensionality reduction is the best when s is set to 1.55 and r is set to 8 for convolutional layers in this paper. And for dense layers, s should be set to 2 and r should be set to 8.

$$\Theta = \{j \mid \text{Mean Rank Std}(j) > s \cup \text{Mean Rank}(j) < r, j \epsilon F\}. \tag{11}$$

The selection of key features by dimensionality reduction rule can be divided into the following two steps:

(1) Select the features with greater MeanRankStd (greater than s)

From the metrics of MeanRankStd, the overall stability of weights ranking can be intuitively judged. Features with smaller MeanRankStd have a relatively stable overall influence on training time, while others with greater MeanRankStd usually fluctuate wildly. As for features with strong ranking fluctuations, we find that they may have a small influence on some input data, but have a large influence on other data. It will cause serious deviations in the prediction of training time without such features. So, we choose to extract features with greater MeanRankStd.

(2) Select the features with smaller MeanRank (smaller than r)

After screening in (1), there are some features with stable rankings (smaller MeanRankStd) in the feature collection. These features contain the ones with smaller MeanRank which have a greater overall impact on training time, such as the feature input channels whose influence is the largest for almost every input data. Therefore, we select the features with smaller MeanRank from the rest of the feature collection.

The process of extracting key features using the dimensionality reduction rule can be described by Algorithm 1.

By using the dimensionality reduction rule, the layer features with no or little impact on training time will be eliminated and the high prediction accuracy will be kept. Therefore, the dimension of layer features and redundant training data for the time prediction model will be reduced. Such as the feature dimension and training data are reduced by 30% and 25% for the convolutional layer, and they are both reduced by 20% for the dense layer. We take Justus et al.'s time prediction model as the baseline to verify the accuracy of our dimensionality reduction rule. Then, we retrain the baseline with the dimension-reduced dataset and compare it with the original baseline. The results show that the prediction error of the model trained with the dimension-reduced dataset remains at the same level as the original baseline. See the experimental analysis in Section 4.3 for details.

Table 7: Weights model vs. baseline.

Datasets	Model	Test RMSE	Test MAPE	Validation RMSE	Validation MAPE
P100_Conv	Baseline	2.962 ms	14.36%	3.455 ms	15.17%
	Weights model	2.894 ms	15.24%	3.091 ms	14.58%
V100_Conv	Baseline	1.722 ms	11.44%	1.547 ms	11.13%
	Weights model	1.660 ms	11.03%	1.850 ms	11.49%
K40_Conv	Baseline	10.136 ms	15.59%	10.361 ms	16.39%
	Weights model	9.051 ms	15.14%	11.675 ms	15.16%
P100_Dense	Baseline	0.033 ms	3.15%	0.032 ms	3.04%
	Weights model	0.032 ms	3.05%	0.032 ms	3.02%
V100_Dense	Baseline	0.051 ms	6.26%	0.046 ms	5.84%
	Weights model	0.046 ms	6.07%	0.047 ms	6.10%
K40_Dense	Baseline	0.157 ms	7.57%	0.179 ms	7.92%
	Weights model	0.159 ms	7.11%	0.150 ms	7.07%
All_Conv	Baseline	4.079 ms	11.44%	4.021 ms	11.16%
	Weights model	4.273 ms	10.40%	3.893 ms	10.39%
All_Dense	Baseline	0.077 ms	4.70%	0.074 ms	4.68%
	Weights model	0.077 ms	4.72%	0.076 ms	4.73%

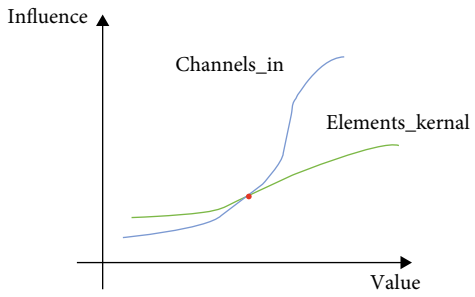

Figure 4: Nonlinear influence.

3.4. Dynamically Setting Number of Local Epochs. In federated learning, each client only uses its own data for model training. And most of the training data are generated by the client itself. Due to the different characteristics of clients, the distribution of data generated by different client is usually different. Therefore, the training data of federated learning is nonindependent and identically distributed (Non-IID). Unfortunately, Non-IID will cause the divergence between the local model and the global model, resulting in the error convergence of the local model. If the number of local epochs set for each client is different, it will undoubtedly aggravate the divergence of the local model. Therefore, FedNova can eliminate the fast error convergence by normalized averaging the gradients of local models and ensure the convergence of the global model when set different number of local epochs for clients. The update of the global model in FedNova can be described by the following formula:

$$x_g^{(t+1)} - x_g^{(t)} = -\tau_{eff}^{(t)} \sum_{i=1}^{m} \frac{n_i}{n} \bullet \eta d_i^{(t)}, \text{ where } \tau_i = \left\lfloor \frac{E \bullet n_i}{B} \right\rfloor. \quad (12)$$

Compared with the update of global model in FedAvg (formula (1)), FedNova uses the normalized averaging gradient $d_i^{(t)}$ to replace the accumulation of the local model's gradient $\sum_{k=0}^{\tau_i - 1} g_i(x_i^{(t,k)})$, which can restrict the error convergence of local models. $\tau_{eff}^{(t)}$ in formula (11) can be treated as the learning rate of the global model. It turns out that compared with FedAvg, FedNova can improve the performance of the global model while reducing the number of communications between the clients and the server.

Although we can set different number of local epochs for clients while ensuring good performance of the global model by using FedNova, the problem of the clients with high training speed waiting for the clients with low training speed still exists. This is because FedNova adopts a random sampling method within a given value range to set the number of local epochs, which may set a small number of epochs for the faster clients and set a large number of epochs for the slower clients.

In order to solve the idling problem and improve the training efficiency of global model, we predict the training time of local models in FedNova for guiding the dynamic setting of local epochs. It should be noted that training time prediction can be combined with any algorithm that supports setting different numbers of local epochs for clients. And we choose FedNova for its excellent performance. We use the time prediction model trained with dimension-reduced datasets to predict the training time of one epoch of local models, and then we calculate the number of local epochs for each client according to the time window of communication between the client and the server. There are many ways to combine FedNova and training time prediction. For example, you can choose to predict the training time of local models on the server side and calculate the number of local epochs, or you can choose to send the time prediction model to clients and let clients calculate the number of local epochs by themselves. In actual scenarios, the

way of dynamically setting the number of local epochs needs to be determined according to specific needs. Next, we will introduce the method based on the FedNova and training time prediction algorithm, and the details are described in Algorithm 2.

First, the server randomly selects some clients according to proportion C and collects their model updates. Second, the server aggregates the updates of the local models to the global model and prepares the time prediction model for clients. Third, the client obtains the global model and time prediction model from the server, divides the local dataset into batches, and extracts the features required for time prediction. Then, the client uses the time prediction model to predict the training time of one epoch of the local model and calculates the number of local epochs according to the preset communication time window T. Finally, the client uses FedNova to update the local model and send it to the server.

4. Experiments

In this section, we arrange experiments to verify the effectiveness of our work, including the weights model and the dimension reduction rule. First, the experimental settings are introduced, including the selection and description of datasets, the setting of model structure, the evaluation metrics, and the introduction of the experimental environment. Second, we use the datasets collected on heterogeneous GPUs to train the weights model for verifying its convergence, and then we conduct a large number of comparative experiments to compare the prediction error between weights model and baseline. We also compare the performance of weights model and linear regression model to prove the superiority of our weights model. Finally, for verifying the effectiveness of the dimensionality reduction rule, we train the baseline based on the complete dataset and the dimension-reduced dataset, respectively, and then compare the error level of prediction between them.

4.1. Experiment Settings

4.1.1. Datasets. We use the layer features described in Section 3.1 and use 8 different datasets for experiments. There are 6 datasets composed of common features, convolutional features, dense features, and training time collected on different GPUs, which we called single-GPU datasets. The other 2 datasets are the stacking of these 6 datasets, which we called stacked datasets. In order to distinguish different types of GPU, we add three hardware features to the stacked datasets: GPU bandwidth, GPU processing units' number, and GPU clock cycle speed. See Table 3 for the description of the datasets.

The layer features selected in the convolutional layer are shown in Table 4. One-hot encoding is used to represent different optimizers and activation functions for more appropriately measuring the feature distance.

The number of features in the dense layer is less than that in the convolutional layer, and some features are different, such as the dimension of input and the dimension of output. The features selected in the dense layer are shown in Table 5.

Table 8: The standard deviation of convolutional feature weights ranking.

Features	P100_ Conv	V100_ Conv	K40_ Conv	MeanRankStd
Batchsize	1.7653	1.3045	1.4026	1.4908
Elements_ matrix	2.3233	2.0775	1.1578	1.8529
Elements_ kernel	1.7904	2.2949	1.4356	1.8403
Channels_in	0.4440	0.4218	0.7599	0.5419
Channels_out	3.0549	2.1915	2.1091	2.4518
Padding	1.6192	1.8036	1.2830	1.5686
Strides	2.4347	2.0012	3.2603	2.5654
Use_bias	1.2580	1.5514	1.3097	1.3730
Opt_SGD	2.3229	3.0940	1.6340	2.3503
Opt_Adadelta	2.2164	2.8144	1.9606	2.3305
Opt_Adagrad	2.7459	2.4789	1.1959	2.1402
Opt_ Momentum	2.6090	2.1952	1.1699	1.9914
Opt_Adam	1.4019	2.1550	2.4311	1.9960
Opt_RMSProp	2.0723	2.3218	0.9707	1.7883
Act_relu	1.5234	1.0455	0.9251	1.1647
Act_tanh	1.7225	1.1762	1.4122	1.4370
Act_sigmoid	1.3042	1.1182	1.1107	1.1777

Table 9: The standard deviation of dense feature weight ranking.

Features	P100_ Dense	V100_ Dense	K40_ Dense	MeanRankStd
Batchsize	1.8927	1.4465	1.5556	1.6316
Dim_input	3.3842	2.2099	2.3296	2.6412
Dim_output	2.4491	2.7088	1.7920	2.3166
Opt_SGD	2.8395	2.9443	2.1840	2.6559
Opt_Adadelta	3.3851	2.9903	1.3891	2.5882
Opt_Adagrad	2.0481	3.0305	2.0840	2.3875
Opt_ Momentum	2.8527	2.9168	1.9885	2.5860
Opt_Adam	2.7488	2.9804	2.4706	2.7333
Opt_ RMSProp	3.4788	1.7232	2.0469	2.4163
Act_relu	1.8273	1.4856	1.0920	1.4683
Act_tanh	2.1289	2.2291	0.7319	1.6966
Act_sigmoid	2.2241	2.1797	1.1457	1.8498

4.1.2. Model Settings. Compared with the baseline, we only add a weights layer at the end of hidden layers for extracting the weights of features, and the rest of the layers are the same as the baseline. Note that the data volume of the stacked datasets (All_Conv and All_Dense) is larger than single-GPU datasets, and three more hardware features are added to distinguish the GPUs. Therefore, for the single-GPU datasets and stacked datasets, the number of weights model's layers are different. Figure 2 shows the settings of the

Table 10: The mean ranking of convolutional feature weights.

Features	P100_ Conv	V100_ Conv	K40_ Conv	MeanRank
Batchsize	8.692	9.574	5.702	7.989333333
Elements_ matrix	5.8396	8.6824	7.5608	7.360933333
Elements_ kernel	11.2952	11.7014	10.1	11.0322
Channels_in	1.09	1.0334	1.0112	1.044866667
Channels_out	7.0608	5.6072	12.0992	8.255733333
Padding	14.1344	13.7096	13.9144	13.91946667
Strides	8.9108	9.8826	8.4852	9.092866667
Use_bias	14.5956	14.595	15.7176	14.9694
Opt_SGD	6.3264	6.853	4.2328	5.804066667
Opt_Adadelta	3.764	7.3048	7.0828	6.050533333
Opt_Adagrad	7.732	5.3538	6.3752	6.487
Opt_ Momentum	5.2476	5.3584	11.0724	7.226133333
Opt_Adam	10.1944	3.6814	5.1448	6.3402
Opt_RMSProp	2.9736	4.107	5.9512	4.343933333
Act_relu	15.8232	15.6576	7.3332	12.938
Act_tanh	14.0576	15.8164	15.514	15.12933333
Act_sigmoid	15.2628	14.082	15.7032	15.016

baseline and weights model for the single-GPU datasets and the stacked datasets.

4.1.3. Metrics. We use the root mean square error (RMSE) and mean absolute percentage error (MAPE) to measure the error between predicted training time and observed training time. The calculation formula of RMSE is shown in equation (13), and the unit is milliseconds (ms); the calculation formula of MAPE is shown in equation (14). $Y_{obs,i}$ represents the observed training time of the i-th input data, and $Y_{pred,i}$ represents the predicted training time of the model of the i-th input data.

$$\mathrm{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}\left(Y_{obs,i} - Y_{pred,i}\right)^2}{n}}, \tag{13}$$

$$\mathrm{MAPE} = \left(\frac{100}{n}\right)\sum_{i=1}^{n}\left|\frac{Y_{obs,i} - Y_{pred,i}}{Y_{obs,i}}\right|. \tag{14}$$

4.1.4. Experimental Environment. The hardware environment of our experiments is a stand-alone machine, using a CPU for training which contains 6 cores and 12 threads, the main frequency of this CPU is 3.1 GHz, and the memory is 16 G. And we use TensorFlow 1.13.2 framework on 64-bit Windows 10 operating system to finish this program.

4.2. Analysis of Weights Model

4.2.1. Convergence Verification of Weights Model. In the convergence verification of weights model, we train the weights model using all datasets. Figure 3 shows the comparison of predicted time and observed time on different datasets. On the *P100_Conv*, *V100_Conv*, *K40_Conv*, and *All_Conv*, the RMSE of weights model is 2.894 ms, 1.660 ms, 9.051 ms, and 4.273 ms, respectively; on the *P100_Dense*, *V100_Dense*, *K40_Dense*, and *All_Dense*, the RMSE of weights model is 0.032 ms, 0.046 ms, 0.159 ms, and 0.077 ms, respectively. Therefore, the weights model has good convergence whether based on single-GPU datasets (*P100*, *V100*, and *K40*) or stacked datasets (*All_Conv* and *All_Dense*).

Table 11: The mean ranking of dense feature weights.

Features	P100_ Dense	V100_ Dense	K40_ Dense	MeanRank
Batchsize	10.3196	10.6268	9.5132	10.1532
Dim_input	6.0088	4.9544	3.854	4.939066667
Dim_output	7.1052	4.882	5.9032	5.963466667
Opt_SGD	4.7872	7.2696	3.9624	5.339733333
Opt_Adadelta	4.396	4.2684	2.2148	3.6264
Opt_Adagrad	5.1912	6.3724	7.1868	6.250133333
Opt_ Momentum	8.072	7.9004	3.562	6.511466667
Opt_Adam	5.4128	3.7576	5.1504	4.7736
Opt_ RMSProp	7.35	2.5372	5.1288	5.005333333
Act_relu	6.0432	10.2896	9.9068	8.746533333
Act_tanh	5.6348	7.3096	11.452	8.132133333
Act_sigmoid	7.6792	7.832	10.1656	8.558933333

4.2.2. Weights Model vs. Linear Regression Model. In order to verify the comparative analysis of the weights model and linear regression model in Section 3.2, we use *P100_ Conv*, *V100_ Conv*, and *K40_ Conv* datasets to train the weights model and linear regression model, respectively. Table 6 shows the RMSE comparison between the linear model and weights model. It can be seen that the error of weights model is far lower than that of the linear regression model. On the *P100_ Conv*, *V100_ Conv*, and *K40_ Conv* datasets, the RMSE of the linear regression model is 9.5 ms, 7.61 ms, and 35.74 ms larger than the weighted model, respectively.

4.2.3. Weights Model vs. Baseline. Since the training time prediction accuracy of the weights model determines the authenticity of feature weights extracted by weights model, we divide the dataset into the training set, test set, and validation set according to the ratio of 8 : 1 : 1 and calculate the test error and the validation error of the weights model to prove the accuracy of weights model. The comparison of test error and verification error between weights model and baseline shows that weights model reaches the same error level as the baseline on *P100*, *V100*, *K40*, and stacked datasets (see Table 7). It is worth mentioning that on the test sets of *K40_Conv*, *K40_Dense*, *P100_Dense*, and *All_Conv*, the MAPE of weights model is lower than baseline by 0.1%, 0.19%, 0.46%, and 1.04%, respectively, and on the test sets

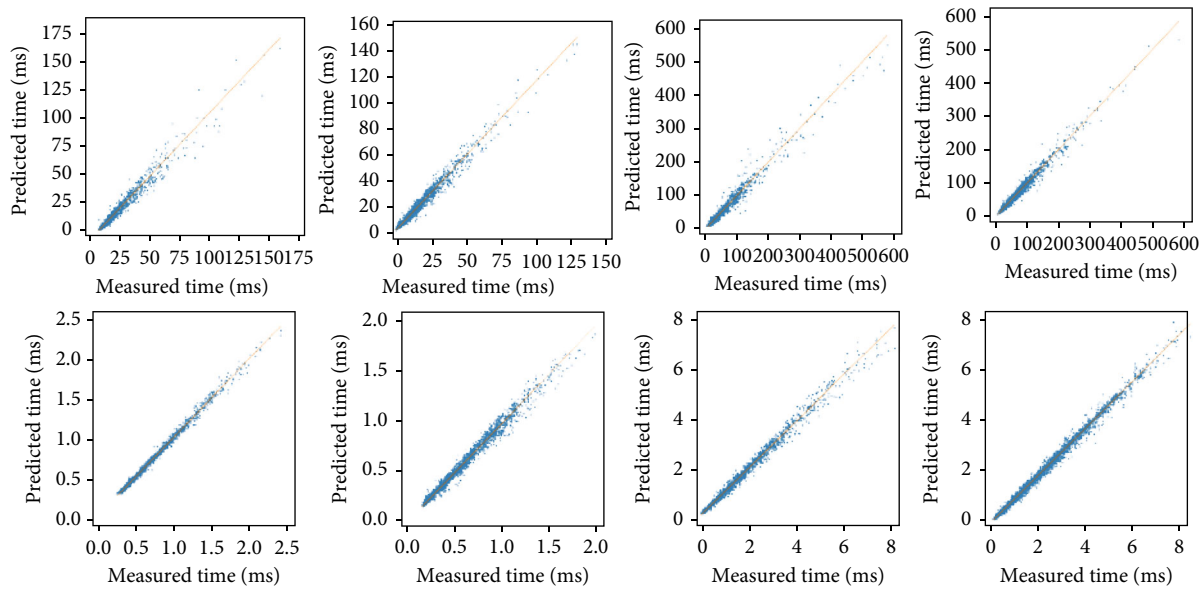

FIGURE 5: Predicted time vs. observed time. The upper of the figure: predicted time vs. observed time on the convolutional feature dimension-reduced datasets, from left to right, is the results on *P100_Conv_small*, *V100_Conv_small*, *K40_Conv_small*, and *All_Conv_small*; the lower of the figure: predicted time vs. observed time on the dense feature dimension-reduced datasets, from left to right, is the results on *P100_Dense_small*, *V100_Dense_small*, *K40_Dense_small*, and *All_Dense_small*.

and validation sets of *P100_Conv*, the RMSE of weights model is lower than baseline by 0.068 ms and 0.364 ms.

4.3. Analysis of Dimensionality Reduction Rule. In 4.2, we proved that the weights model has good convergence and has a high prediction accuracy that is not lower than baseline. In this section, we will verify the effectiveness of the dimensionality reduction rule. We use the weights model to extract feature weights of different test sets to form the weights datasets, then we use the dimensionality reduction rule described in Section 3.3 to reduce the dimension of features. After that, we compare the prediction error of the model trained based on dimensionality-reduced data with the original baseline.

In our experiments, we found that the ranking of feature weights may fluctuate significantly for different input feature data. We think it is because the influence of layer features on training time does not necessarily change linearly. Assuming the influence of *channels_in* and *elements_kernel* on the final training time is shown in Figure 4, when the value of *channels_in* gradually decreases, its influence on the final result is not necessarily greater than the *elements_kernel*.

Before using the dimensionality reduction rule, we use formula (8) to calculate the ranking of feature weights in each weights dataset. According to the first step of the dimensionality reduction rule, we need to measure the fluctuation of each feature weight, that is, the standard deviation. Therefore, the standard deviation of weights ranking and the average standard deviation of weights ranking are calculated, see Tables 8 and 9. For the second step of the dimensionality reduction rule, we calculate the average ranking of each feature's weight on all weights datasets and obtain the overall average ranking on all datasets according to formula (9). Tables 10 and 11, respectively, show the average ranking of feature weights on each dataset, as well as the overall average ranking MeanRank.

Taking the convolutional features as an example, since the optimizers and activation functions are represented by one-hot encoding, we regard all optimizer fields (feature name starting with *opt_*) as one feature *opt* and all activation function fields (feature name starting with *act_*) as one feature *act*.

According to step 1 of the dimensionality reduction rule, we select features with MeanRankStd greater than 1.55, the results are *elements_matrix*, *elements_kernel*, *channels_in*, *channels_out*, *strides*, and *opt* (all optimizer fields). The remaining features with small MeanRankStd are *batchsize*, *channels_in*, *padding*, *use_bias*, and *act* (all activation fields).

According to step 2 of the dimensionality reduction rule, we select features with MeanRank less than 8 in *batchsize*, *channels*, *padding*, *use_bias*, and *act*, and the results are *batchsize* and *channels_in*.

Finally, by using the dimensionality reduction rule, we get *batchsize*, *channels_in*, *elements_matrix*, *elements_kernel*, *channels_in*, *channels_out*, *strides*, and *opt*. The convolutional layer features are reduced by 3 dimensions (*padding*, *use_bias*, and *act*), and the training data is reduced by 5 dimensions (*padding*, *use_bias*, *act_relu*, *act_tanh*, and *act_sigmoid*).

For the dense layer features, we use the same method. According to the dimensionality reduction rule, we exclude features that have less influence on training time, including *act_relu*, *act_tanh*, *act_sigmoid*, and *batchsize*. It should be noted that the dense layer has the parameter-intensive characteristic, which means that the transmission of parameters takes a long time. However, the time overhead of parameter transmission was ignored in our datasets. According to the forward propagation process of the neural network, the

Table 12: Baseline vs. baseline_small.

Datasets	Model	Test RMSE	Test MAPE	Validation RMSE	Validation MAPE
P100_Conv	Baseline	2.962 ms	14.36%	3.455 ms	15.17%
	Baseline_small	2.838 ms	15.27%	3.0948 ms	15.47%
V100_Conv	Baseline	1.722 ms	11.44%	1.547 ms	11.13%
	Baseline_small	1.762 ms	11.80%	1.791 ms	11.80%
K40_Conv	Baseline	10.136 ms	15.59%	10.361 ms	16.39%
	Baseline_small	9.882 ms	16.01%	11.508 ms	16.68%
P100_Dense	Baseline	0.033 ms	3.15%	0.032 ms	3.04%
	Baseline_small	0.027 ms	2.86%	0.031 ms	2.92%
V100_Dense	Baseline	0.051 ms	6.26%	0.046 ms	5.84%
	Baseline_small	0.044 ms	5.49%	0.045 ms	5.45%
K40_Dense	Baseline	0.157 ms	7.57%	0.179 ms	7.92%
	Baseline_small	0.154 ms	6.60%	0.140 ms	6.60%
All_Conv	Baseline	4.079 ms	11.44%	4.021 ms	11.16%
	Baseline_small	4.001 ms	11.89%	4.090 ms	12.10%
All_Dense	Baseline	0.077 ms	4.70%	0.074 ms	4.68%
	Baseline_small	0.069 ms	4.70%	0.070 ms	4.75%

dense layer must perform one forward propagation calculation for one input data, and it must perform a batch of forward propagation for a batch of input data. Therefore, the batchsize determines the number of times the parameters are transmitted, which should not be excluded.

After extracting key features by dimensionality reduction rule, we filter the training sets and get the dimension-reduced datasets on *P100_Conv*, *V100_Conv*, *K40_Conv*, *All_Conv*, *P100_Dense*, *V100_Dense*, *K40_Dense*, and *All_Dense* (We add _ *small* represents the dimension-reduced datasets. For example, the dimension-reduced dataset of *P100_Conv* is *P100_ Conv_ small*). For verifying the validity of dimension-reduced datasets, we use the dimension-reduced datasets to train the baseline, and the trained model is called baseline_small. And our experiments have proved that baseline_small also has good convergence. Figure 5 shows the comparison of predicted time and observed time of baseline_small on the test sets.

For determining whether the dimension-reduced datasets caused the loss of prediction accuracy, we calculated the RMSE and MAPE of baseline and baseline_small on all datasets. The results are shown in Table 12. For the convolutional layer datasets, the baseline_small test RMSE is 0.1385 ms lower than the baseline on average, the verification RMSE is 0.3664 ms higher than the baseline on average; for the dense layer datasets, the baseline_small test RMSE is 0.0078 ms lower than the baseline on average, and the verification RMSE is 0.015 ms lower than the baseline. It is worth mentioning that for the dense layer datasets, both RMSE and MAPE of baseline_small are lower than baseline. We consider it is because there are some features in the dense layer that have no contribution to the training time. These features will disturb the data distribution and increase the prediction error of the model.

The experiments show that after reducing the convolutional layer features by 30% and the training data by 25%, the error level of prediction is still consistent with the original baseline; after reducing the dense layer features by 20% and the training data by 20%, the error level is generally lower than the baseline, which proves the effectiveness of our weights model and dimensionality reduction rule.

5. Conclusion

For the problem of setting the number of local epochs for heterogeneous clients in federated learning, we propose a solution of predicting the training time of deep learning tasks on clients to guide the dynamic setting number of local epochs. We design the weights model to extract the weights of features and accurately interpret the relationship between model features and training time. This paper focuses on the combination of weights model and dimensionality reduction rule to extract the key features for reducing the dimension of features and redundant training data required by the time prediction model. The purpose of our work is to improve the feasibility of predicting the training time of deep learning models on heterogeneous clients in federated learning, so as to dynamically set the number of local epochs for clients. Compared with the existing methods, the results of our experiments show that (1) the weights model has good convergence on heterogeneous devices; (2) the predicted training time of weights model reaches the same error level as baseline; (3) the dimensionality reduction rule in this paper can reduce 30% features and 25% redundant data for the convolutional layer and reduce 20% features and 20% redundant data for the dense layer, while maintaining high prediction accuracy.

Data Availability

Previously reported [Model Features] data were used to support this study and are available at [10.1109/

BigData.2018.8622396]. These prior studies (and datasets) are cited at relevant places within the text as references [8].

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant no. 62072146 and no. 61972358 and the Key Research and Development Program of Zhejiang Province (2019C01059, 2019C03135, and 2019C03134).

References

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," in *Artificial intelligence and statistics*, pp. 1273–1282, PMLR, 2017.

[2] A. Hard, K. Rao, R. Mathews et al., "Federated learning for mobile keyboard prediction," 2018, https://arxiv.org/abs/1811.03604.

[3] "ai.google. Under the hood of the Pixel 2: How AI is supercharging hardware, 2018," 2018, https://ai.google/stories/ai-in-hardware/.

[4] "support.google. Your chats stay private while Messages improves suggestions, 2019," 2019, http://support.google.com/messages/answer/9327902.

[5] Apple, "Private federated learning (NeurIPS 2019 Expo Talk Abstract)," 2019, https://nips.cc/ExpoConferences/2019/schedule?talk_id=40.

[6] T. Li, A. K. Sahu, M. Zaheer, M. Sanjabi, A. Talwalkar, and V. Smith, "Federated optimization in heterogeneous networks," 2018, https://arxiv.org/abs/1812.06127.

[7] J. Wang, Q. Liu, H. Liang, G. Joshi, and H. V. Poor, *Tackling the Objective Inconsistency Problem in Heterogeneous Federated Optimization*, 2020.

[8] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big data)*, Seattle, WA, USA, December 2018.

[9] E. R. Edelman, S. M. J. van Kuijk, A. Hamaekers, M. J. M. de Korte, G. G. van Merode, and W. F. F. A. Buhre, "Improving the prediction of total surgical procedure time using linear regression modeling," *Frontiers in Medicine*, vol. 4, p. 85, 2017.

[10] B. Yu, H. Wang, W. Shan, and B. Yao, "Prediction of bus travel time using random forests based on near neighbors," *Computer-Aided Civil and Infrastructure Engineering*, vol. 33, no. 4, pp. 333–350, 2018.

[11] J. Cheng, G. Li, and X. Chen, "Research on travel time prediction model of freeway based on gradient boosting decision tree," *IEEE Access*, vol. 7, pp. 7466–7480, 2019.

[12] Q. Hang, E. R. Sparks, and A. Talwalkar, *Paleo: A Performance Model for Deep Neural Networks*, 2016.

[13] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, *Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters*, 2018.

[14] W. Xu, H. Peng, X. Zeng, F. Zhou, X. Tian, and X. Peng, "A hybrid modelling method for time series forecasting based on a linear regression model and deep learning," *Applied Intelligence*, vol. 49, no. 8, pp. 3002–3015, 2019.

[15] D. Velasco-Montero, J. Fernandez-Berni, R. Carmona-Galan, and A. Rodriguez-Vazquez, "PreVIous: a methodology for prediction of visual inference performance on IoT devices," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9227–9240, 2019.

[16] N. C. Petersen, F. Rodrigues, and F. C. Pereira, "Multi-output bus travel time prediction with convolutional LSTM neural network," *Expert Systems with Applications*, vol. 120, pp. 426–435, 2019.

[17] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *Computer Science*, 2015, https://arxiv.org/abs/1409.1556.

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, July 2016.

[19] R. Adolf, S. Rama, B. Reagen, G.-y. Wei, and D. Brooks, "Fathom: reference workloads for modern deep learning methods," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Providence, RI, USA, September 2016.

[20] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating deep network training by reducing internal covariate shift," *JMLR*, 2015, https://arxiv.org/abs/1502.03167.