

A Model Distributed Training Method Based on Improved Split Learning in the Internet of Things

Hui Cai, Meifeng Zhu, Shiji Tao, Yuning Zhou, Jian Zhou*, Biyun Sheng, Lu Zhao, Xin He

Nanjing University of Posts and Telecommunications, Nanjing 210023, China

Keywords: Deep echo state network; Distributed training; Split learning

Citation: Cai H., Zhu M.F., Tao S.J., et al.: A Model Distributed Training Method Based on Improved Split Learning in the Internet of Things. *Data Intelligence*, 2026, Vol. 8, Art. No.: 2026r03. DOI: <https://doi.org/10.3724/2096-7004.di.2026.0040>

ABSTRACT

With the rapid development of the Internet of Things (IoT), a large amount of data has been brought to distributed nodes in edge environments, which has caused concerns about data privacy leakage and data transmission latency. Split learning (SL) divides the model among distributed nodes, addressing the high storage and computational demands brought by storing the entire model on a single node in federated learning. However, SL suffers from scalability limitations, leading to decreased training effectiveness with a changing number of nodes in IoT. Furthermore, the heterogeneity of data in IoT nodes also affects the training effectiveness due to the generalization limitations of SL. Therefore, we propose a model distributed training method based on improved SL in IoT. Firstly, we leverage Mixup, which is a data augmentation method to blend the “smashed data” generated by client models, thereby creating more diverse samples with smooth transitions in the feature space. This improves robustness against the influence of node number fluctuations. Secondly, a hypernetwork is employed to allocate weights to model parameters, obtaining personalized training parameters for each client model by evaluating the importance of different clients, thus improving the impact of heterogeneous node data on model generalization performance. Experimental results show that the proposed method outperforms other distributed training methods on various datasets in terms of model performance.

1. INTRODUCTION

With the rapid development of Internet of Things (IoT) [1], IoT devices generate an increasing amount of data. Deep learning models [2], typically consisting of multiple hierarchical layers, are capable of quickly extracting sufficient features from vast amounts of data and have been applied to tasks related to

*Corresponding author: Jian Zhou (E-mail: zhoujian@njupt.edu.cn; ORCID: 0009-0003-9201-8098).

IoT. However, transmitting the massive data generated by IoT devices to a centralized location for deep learning model training poses challenges such as data transmission latency and data privacy leakage. When current edge distributed nodes gradually possess higher computational capabilities, deep learning models can be deployed on edge nodes near IoT devices for further distributed training, mitigating the aforementioned challenges [3].

Federated Learning (FL) [4] is an emerging distributed training method consisting of multiple distributed nodes and a central server. Nodes utilize local data for model training, and the central server updates global model parameters based on the parameters uploaded by distributed nodes. FL avoids the transmission latency of raw data and privacy leakage issues in deep learning model training. However, it requires placing and training deep learning models on each node, typically demanding high storage and computational performance for individual nodes. Hence, Split Learning (SL) [5], proposed as an improved distributed training method for FL, splits the model on individual nodes into client-side and server-side models, which largely decreases computational and storage burden.

SL achieves secure collaboration by breaking down models without the need to share raw data, only intermediate calculation results or model parameters. The split layers in SL are typically positioned as close to the input layer as possible. The client-side model, which handles the less computationally intensive portion of the deep learning model, is deployed across distributed nodes, while the more computationally intensive portion, which resides on the server-side model, is deployed in the cloud. In SL, the client-side models on nodes use local data as input to generate representations of split layers, i.e., smashed data. Subsequently, nodes send these smashed data to the cloud, where the server-side model continues training them to generate global model update parameters. On the one hand, SL offloads most of the computation to the cloud, effectively reducing the storage and computational demands of deep learning model training for individual distributed nodes. On the other hand, SL only involves the transmission of intermediate representations of the model without exposing raw data, further reducing data transmission latency and privacy leakage.

In IoT, the number of distributed nodes often fluctuates due to network link failures and recovery, the depletion and replenishment of node energy. Considering the training of the serverside model in SL relies on the client-side model, the training results of the client-side model will become unstable with the change of the number of nodes, thereby affecting the overall model training, namely, the non-scalability of SL nodes. Currently, some work combines FL with SL. After the completion of back propagation in SL, FL is applied across clients to alleviate the impact of fluctuations in client numbers on the server side through aggregation [6]. While other work [7] conducts local model parallel training on clients after the completion of back propagation in SL, to address the impact of client number fluctuations on overall model training. However, the above work fails to consider the issue of client model separation during the forward propagation process in SL, so the scalability remains relatively weak.

Moreover, since distributed nodes in the IoT are deployed in fixed locations, collected data by distributed nodes are typically non-independent and non-identically distributed (non-IID). The adoption

of non-IID data as input for SL and conducting deep learning model training can significantly affect the results of SL model training, namely, the generalizability of SL. Previous work [8] uses data-driven methods to reduce statistical differences among client datasets, but this may pose risks of privacy leakage. Other work [9-10] employs model-based methods to customize personalized model structures or parameters for different clients, but the scale used to describe the similarity between models may be inaccurate, resulting in suboptimal model performance.

Mixup [11] generates new samples and labels by linearly interpolating between data and labels, which enhances the performance of deep learning models. Specifically, for two randomly selected samples and labels, Mixup blends them according to a certain ratio to create a new sample and label, where the new sample and label are the weighted average of the original samples and the original labels, respectively. Mixup helps alleviate the sensitivity of models to noise and variations. Therefore, we utilize Mixup to blend smashed data and corresponding labels to address the scalability issues of SL. Hypernetworks (HNs) [12] generate parameters for other neural networks by mapping embeddings of the target task to corresponding model parameters, enabling the assessment of importance for each layer of the model. Therefore, we utilize HNs to explore finer-grained personalized strategies to adapt to non-independent and identically distributed clients.

We conducted relevant experiments on different datasets and deep learning models, comparing the proposed Mixup and HNs-based improved split learning method with other distributed learning methods. The experimental results show that our method has superior generalizability and model performance in a distributed IoT environment. We summarize our main contributions as follows.

- To address the impact of changes in the number of nodes in IoT on model training, we propose a smashed data mixing method based on Mixup [11]. This method generates more diverse features by mixing smashed data, thereby enhancing robustness to variations in node count.
- To address the problem of data heterogeneity among nodes in the IoT, we propose a weight allocation method based on HNs [12]. We assign appropriate weight coefficients to each client model using HNs on the server side, thereby generating new models that adapt to the diverse data of different clients.
- We conduct relevant experiments and show that the proposed method demonstrates superior generalization and model performance in a distributed IoT environment compared to other distributed training methods.

The remaining sections of our paper are organized as follows. Section 2 presents related work. Section 3 and Section 4 propose our method. Section 5 analyzes the simulation results. Section 6 summarizes our paper.

2. RELATED WORK

Due to the ability to perform highly accurate and reliable inference tasks, deep learning models have been widely applied in fields such as image recognition and natural language processing in IoT. For example, Su et al. [13] utilized convolutional neural networks for lightweight classification of IoT malware based on image recognition, while Breve et al. [14] employed natural language processing models to trigger security and privacy violation rules in IoT platforms. However, deep learning models typically require significant computational resources, which are not well supported by resource-constrained IoT devices.

FL is a distributed training method that utilizes decentralized data for model training. The FedAvg algorithm, proposed by McMahan et al. [15], is a fundamental FL algorithm. FedAvg learns a global model by aggregating local models trained on each distributed node, thus avoiding privacy breaches associated with transmitting raw local data. Wang et al. [16] proposed Federated Training over Encrypted Data (FedTOE), which optimizes transmission latency constraints by jointly allocating wireless resources and quantizing among clients in non-ideal wireless channel environments. The SCAFFOLD algorithm, introduced by Karimireddy et al. [17], enhances communication efficiency by maintaining a control variable adjusted based on global model gradient changes in each round, making communication more compact among devices. Wang et al. [18] proposed the In-Edge AI framework, which integrates FL based on deep reinforcement learning with mobile edge computing systems to optimize resource allocation. Kang et al. [19] proposed Reliable Federated Learning in Mobile Networks, which improves the reliability of federated learning tasks by using the concept of “reputation” as a measure to select reliable data for training. However, while these methods optimize various aspects of the FL process such as transmission latency, privacy leakage, and training inefficiency, the resources of distributed nodes are often limited, making it difficult to support running deep learning models on clients.

SL is an improved FL model distributed training method suitable for resource-constrained distributed nodes. Gupta et al. [5] first introduced the concept of split learning, where nodes jointly train a complete model without leaking raw data while distributing the computationally intensive parts of the model to high-powered nodes. The SplitNN distributed training method proposed by Vepakomma et al. [20] significantly improves resource efficiency by providing different training modes without sharing raw data and model details with other distributed nodes. Samikwa et al. [21] introduced Adaptive Resource-aware Split-learning (ARES) based on the IoT, which minimizes the impact of lagging nodes on training by utilizing device segmentation, considering time-varying network throughput and computational resources, thus effectively improving model training speed and reducing training energy consumption. Zhou et al. [22] proposed a multi-hop split learning method (M-SLRNN) for Recurrent Neural Networks (RNNs) [23] in mobile edge computing. M-SLRNN splits the RNN into multiple sub-models and places them on a group of mobile devices, exchanging intermediate results only through device-to-device multi-hop links, effectively improving model training efficiency. Thapa et al. [24] proposed a hybrid method called SplitFed, which combines the updated pattern of FL with the model splitting of SL to reduce model training time while lowering resource requirements for distributed nodes. However, the scalability and generalizability

of SL have yet to be improved in the above methods. Therefore, this paper proposes an improved SL to achieve efficient model distributed training in IoT.

Recent advances further strengthen the context of split learning and federated learning in IoT. For communication and system-aware SL, Ninkovic et al. [25] proposed COMSPLIT, a communication-aware split learning design tailored to heterogeneous IoT channel conditions. For personalization federated learning under data heterogeneity, hypernetwork-based PFL methods have shown notable progress. In particular, PeFLL [26] leveraged hypernetworks in a learning-to-learn manner, while MH-pFedHN [27] explored hypernetwork-driven personalization for model-heterogeneous clients. These studies highlight the growing importance of combining communication efficiency and personalization in future IoT-oriented learning systems.

3. SYSTEM MODEL

Assuming N distributed nodes and one cloud server in IoT, less computationally intensive client-side models are deployed across distributed nodes. In contrast, more demanding server-side models are deployed in the cloud. The distributed model training architecture based on improved SL is illustrated in Figure 1.

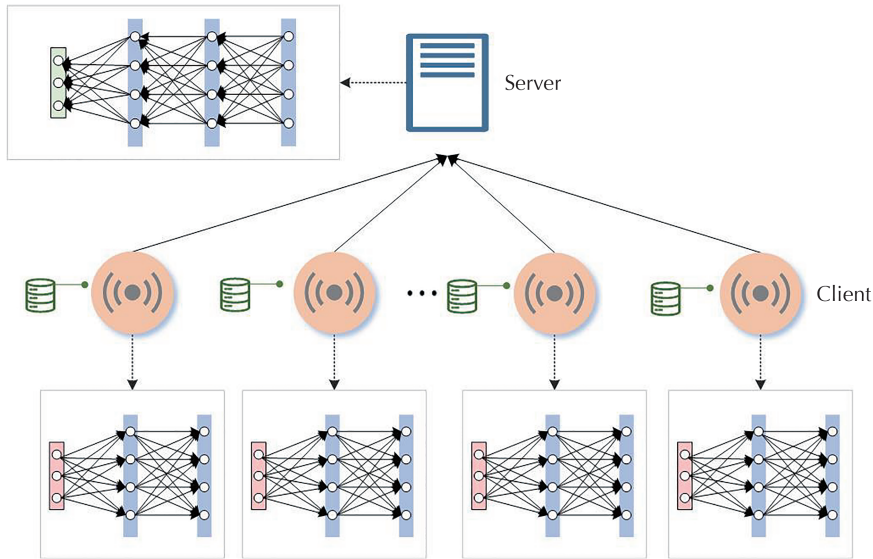


Figure 1. Architecture of Model Distributed Training based on improved SL.

The training process is detailed as follows. Firstly, the input data is denoted as $U = \{u_1, u_2, \dots, u_N\}$. The input data undergoes training by client-side models to generate smashed data at the cutting layer, denoted as $x_i = f(u_i)$, where $f(\cdot)$ is the activation function. After obtaining the smashed data at the cutting layer, it is

uploaded to the server-side. Then, on the server-side, on the one hand, the method for mixed-up smashed data based on Mixup [11] is utilized. Mixed-up smashed data and mixed labels are obtained to present a smooth transition in the feature space and improve robustness to changes. On the other hand, the method for weight allocation based on HNs [12] is utilized. Weights are assigned to the deep learning model parameters trained on different client models during the model training process using smashed data and mixed-up smashed data on the server side. Model layer parameters with higher correlation are assigned higher weight-matching coefficients to improve the impact of data heterogeneity on model training.

For clarity, all notations used in this paper are summarized in Table 1.

Table 1. Notation Table.

Symbol	Description
δ	the output of HNs
ψ	the parameter of HNs
λ	the probability parameter
$\mathbb{B}$	the training data set
$C_i^{N_s}$	the subset obtained by sampling $N - 1$ nodes without node i
$f(\cdot)$	the activation function
x_i	the smashed data
$x_{i,j}^{i',j'}$	the mixed-up smashed data
$y_{i,j}^{i',j'}$	the mixed label
$\tilde{L}_{i,j}^{i',j'}(\cdot)$	the loss function of the single mixed-up smashed data
$\tilde{L}_i^{i'}(\cdot)$	the loss function of mixed-up smashed data for client i and i'
$\tilde{L}_i(\cdot)$	the loss function of the i -th client's mixed-up smashed data
$L_i(\cdot)$	the objective loss function of the client i
$\bar{L}_i(\cdot)$	the loss function of training smashed data on the server
N	the number clients
U	the input data
V	the input of HNs
W	the model parameter
W^l	the model parameters of layer l
$\overline{W}_i$	the new model parameter of client i by weighting every client's W according to δ_i

4. THE DESIGN OF MHSL

4.1 Overview

The model training process of MHSL is illustrated in Figure 2. Firstly, after receiving the smashed data from the client, the server uses Mixup to mix the smashed data and its labels through linear interpolation, resulting in new mixed-up smashed data and mixed-up labels. Afterward, the smashed data and mixed-up smashed data are used as inputs for training the client model. During the training process, embedding vectors are used as inputs to HNs to evaluate the parameter weights of each layer in each model, where higher weight ratio coefficients are assigned to model layer parameters with higher correlation. Finally, the weight ratio coefficient is weighted with the model parameters to obtain the personalized model parameters corresponding to each client.

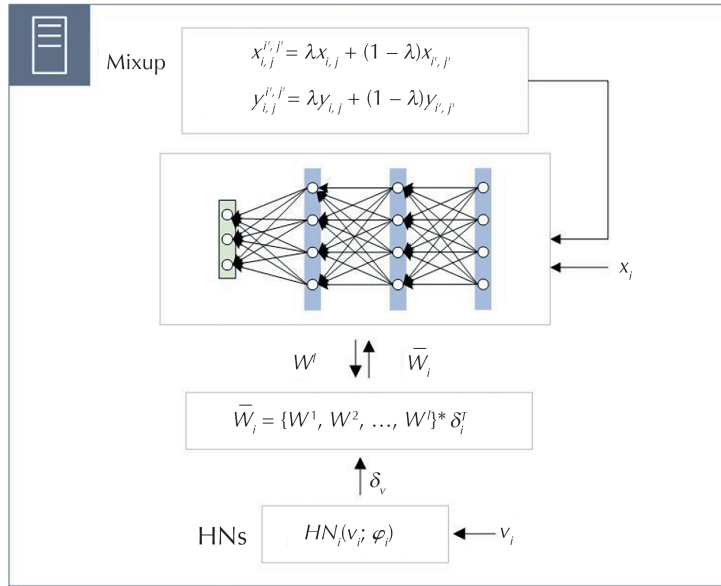


Figure 2. The Model Training Process of MHSL.

4.2 Smashed data mixing method based on Mixup

Inspired by previous work [11], Mixup blends feature information in the sample space, generating more diverse new samples and presenting a smoother transition in the feature space, which can increase robustness to variations. Therefore, in this section, Mixup is applied to blend smashed data and their labels on each distributed node, addressing the scalability issues of SL limited by fluctuations in IoT node number.

The specific process of the smashed data mixing method based on Mixup is as follows. Firstly, considering only two clients, denoted as i and i' , after all smashed data and their corresponding labels are

uploaded to the server-side, the server-side mixes the j -th smashed data from client i and the j' -th smashed data from client i' to generate mixed-up smashed data, and mixes the j -th label from client i and the j' -th label from client i' to generate mixed-up labels. It can mix features in the sample space, generating new samples with a certain diversity, effectively increasing the model's sensitivity to noise and variations. The mixing formula is as follows:

$$\begin{aligned} x_{i,j}^{i',j'} &= \lambda x_{i,j} + (1-\lambda)x_{i',j'} \\ y_{i,j}^{i',j'} &= \lambda y_{i,j} + (1-\lambda)y_{i',j'} \end{aligned} \quad (1)$$

where the probability parameter λ follows a Beta distribution, i.e., $\lambda \sim (\alpha, \beta)$. When $\alpha = \beta = 1$, λ follows a uniform distribution.

After the smashed data is mixed, model training is conducted on the server-side using the mixed-up smashed data $x_{i,j}^{i',j'}$ and mixed labels $y_{i,j}^{i',j'}$. $x_{i,j}^{i',j'}$ is served as the input to the server-side model, while $y_{i,j}^{i',j'}$ is served as the corresponding expected output. For each mixed-up smashed data, its loss function is denoted as $\tilde{L}_{i,j}^{i',j'}(W)$. Similarly, considering only two clients i and i' , the loss function generated by the mixture of the i -th and i' -th clients' smashed data is $\tilde{L}_i'(W) = 1/b \sum_{j \in \mathbb{B}_i, j' \in \mathbb{B}_{i'}} \tilde{L}_{i,j}^{i',j'}(W)$, where b represents the size of the client's training data, and $\mathbb{B}_i$ and $\mathbb{B}_{i'}$ represent the training data sets.

For N clients, each smashed data is mixed with the smashed data from other clients, resulting in $N - 1$ loss functions for each client. Let C_i^{Ns} denote the subset obtained by sampling $N - 1$ elements from all nodes except node i without replacement. The loss function for the i -th client's mixed-up smashed data can be represented as $\tilde{L}_i(W) = \sum_{i' \in C_i^{Ns}} \tilde{L}_i^{i'}(W)$. Then, this loss function is used as a regularization term for the final objective loss function to improve the non-scalability of SL nodes. The objective loss function is denoted as

$$L_i(W) = \underset{W}{\operatorname{argmin}} \sum_i \tilde{L}_i(W) + \tilde{L}_i(W) \quad (2)$$

where $\tilde{L}_i(W)$ represents the loss function for the continued training of smashed data at the server.

4.3 Weight Allocation Method based on HNs

HNs are a type of deep neural network that dynamically outputs the weights of input parameters during training tasks [12]. HNs maintain a personalized model for each client on the server-side. HNs do not require transmitting redundant parameters, thus avoiding additional communication costs. Therefore, we use HNs to train the server-side model parameters to obtain weight allocation coefficients. These coefficients are then assigned to the model parameters to address the generalization issue of SL limited by data heterogeneity among IoT nodes.

HNs consist of several fully connected layers. The parameters of HNs are denoted as $\psi = \{\varphi_1, \dots, \varphi_i, \dots, \varphi_N\}$, where ψ represents the set of parameters. The input to HNs is an embedding

vector $V = \{v_1, \dots, v_{i'}, \dots, v_N\}$, which is automatically updated with model parameters. The output is a weight matrix $\delta = \{\delta_1, \dots, \delta_{i'}, \dots, \delta_N\}$. The weight matrix for the i th client is defined as:

$$\delta_i = HN_i(v_i; \phi_i) \quad (3)$$

where v_i is the embedding vector for client i , ϕ_i represents the parameters of the HNs for client i , and $HN_i(\cdot)$ denotes the corresponding HN for client i .

The specific process of the weight allocation method based on HNs is as follows. The server receives the smashed data x_i uploaded by client i as the input for training the server-side model. Let W_i be the model parameters corresponding to client i after training. We obtain the new model parameters $\bar{W}_i$ by weighting each layer parameter in W_i according to δ_i :

$$\bar{W}_i = \{\bar{W}_i^1, \dots, \bar{W}_i^l\} = \{W^1, W^2, \dots, W^l\} * \delta_i \quad (4)$$

where l is the number of layers in the deep learning model. Then, the loss function corresponding to client i on the server side is:

$$L_i(\bar{W}_i) = \underset{\bar{W}_i}{\operatorname{argmin}} \bar{L}_i(\bar{W}_i) + \tilde{L}_i(\bar{W}_i) \quad (5)$$

where $\tilde{L}_i(\bar{W}_i)$ is the loss term corresponding to the mixed-up smashed data and mixed-up labels after Mixup mixing. $\bar{L}_i(\bar{W}_i)$ is the loss function corresponding to the weighted proportion coefficients assigned by HN_i . Using HNs to calculate the proportion coefficients alleviates the impact of heterogeneous data on DESN training by considering the weights of the output weight matrices of all clients i .

Next, we introduce the update rules for V and ψ . From the above equation, the gradients concerning v_i and ϕ_i are obtained as:

$$\begin{aligned} \nabla_{v_i} L_i(\bar{W}_i) &= (\nabla_{v_i} \bar{W}_i)^T \nabla_{\bar{W}_i} L_i(\bar{W}_i) = \\ &= \left[\{W^1, W^2, \dots, W^l\} * \nabla_{v_i} HN_i(v_i; \phi_i) \right]^T \nabla_{\bar{W}_i} L_i(\bar{W}_i) \end{aligned} \quad (6)$$

$$\begin{aligned} \nabla_{\phi_i} L_i(\bar{W}_i) &= (\nabla_{\phi_i} \bar{W}_i)^T \nabla_{\bar{W}_i} L_i(\bar{W}_i) = \\ &= \left[\{W^1, W^2, \dots, W^l\} * \nabla_{\phi_i} HN_i(v_i; \phi_i) \right]^T \nabla_{\bar{W}_i} L_i(\bar{W}_i) \end{aligned} \quad (7)$$

where $\nabla_{\bar{W}_i} L_i(\bar{W}_i)$ can be obtained from the local training of client i in each update. Since $\nabla_{v_i} HN_i(v_i; \phi_i)$ and $\nabla_{\phi_i} HN_i(v_i; \phi_i)$ are gradients of δ_i in the directions of v_i and ϕ_i , v_i and ϕ_i can be updated in a general way as follows:

Algorithm 1. A Model Distributed Training Algorithm based on improved Split Learning (MHSL).

Require: input $U_i = \{U_i^1, U_i^2, \dots, U_i^l\}$, expected output Y

Ensure: Weight matrix $\bar{W}_i$

1: **Client:**

2: for each mini-batch U_i^j do

3: Each client i performs client model training to obtain smashed data;

4: Upload smashed data and labels to the server-side;

5: **Server:**

6: for $k = 1, 2, \dots, m$ do

7: $\bar{W}_i(k+1) = \{W^1, W^2, \dots, W^l\} * \nabla_{v_i} HN_i(v_i(k); \phi_i(k))$

8: $\Delta W_i \leftarrow$ Update $\bar{W}_i(k+1)$;

9: Update $W^1, W^2, \dots, W^l$, based on ΔW_i

10: Update $v_i(k+1)$ and $\phi_i(k+1)$

11: end for

12: Update $\bar{W}_i(k+1)$:

13: for each mini-batch $U_i^j \in U_i$ do

14: Implement the Mixup-based method for mixing smashed data, blending smashed data and labels.

15: Utilize the mixed-up smashed data to continue server-side model training.

16: Utilize the smashed data to continue executing server-side model training, incorporating the mixed-up smashed data as a regularization term to obtain the final objective loss function, and compute W_{i^*}

17: $\Delta W_i = W_{i^*} - \bar{W}_i(k+1)$

18: **return** $\bar{W}_i$

$$\Delta v_i = (\nabla_{v_i} \bar{W}_i)^T \Delta W_i = \left[\{W^1, W^2, \dots, W^l\} * \nabla_{v_i} HN_i(v_i; \phi_i) \right]^T \Delta W_i \quad (8)$$

$$\Delta\varphi_i = (\nabla_{\varphi_i} \overline{W}_i)^T \Delta W_i = \left[\{W^1, W^2, \dots, W^I\} * \nabla_{\varphi_i} HN_i(v_i; \varphi_i) \right]^T \Delta W_i \quad (9)$$

According to the above formulas, v_i and φ_i can be updated in each communication round, and then the weight matrix δ_i can be updated.

The algorithm flowchart is shown in Algorithm 1. N distributed nodes and one cloud server are deployed.

We next discuss communication and computation costs of MHSL. Specifically, Mixup only incurs lightweight linear interpolation on smashed data, where computational costs are proportional to each client's sample numbers. In addition, HyperNetworks mainly produces computational costs from running algorithms on additional fully connected layers on the server side. The communication overhead mainly derives from the transmission for the smashed data uploaded by each client.

5. SIMULATION AND ANALYSIS

Python is used as the simulation tool to validate the effectiveness of MHSL. The CNN, LSTM, and DESN are separately simulated and analyzed on MNIST [28], FEMNIST [29], and CIFAR-10 [30] datasets. Firstly, the simulation settings are introduced. Then, The datasets and comparative methods are described. Next, the performance of the proposed method on different datasets are evaluated. Finally, the scalability and generalization of the proposed method are verify by ablation experiments.

5.1 Simulation Settings

Unless otherwise noted, we set the learning rate as 0.1 with the Adam optimizer, use a batch size of 128, and train for 200 epochs. In IoT, 10 to 50 distributed nodes and a cloud end are deployed. We divide each deep learning model into client-side and server-side models. Specifically, the input layer, one convolutional layer, and one pooling layer of CNN are placed on the client-side, while one convolutional layer, one pooling layer, two fully connected layers, and the output layer are placed on the server-side. For LSTM, the input layer and the first hidden layer of LSTM are placed on the client-side, with the remaining hidden layers and output layer is placed on the server-side. For DESN, the input layer and the first reservoir pool of DESN are placed on the client-side, with the remaining reservoir pools and output layer are placed on the server-side. The specific parameter settings for CNN, LSTM, and DESN are shown in Table 2.

Table 2. Model Parameters.

Model	Parameters	Value
CNN	Number of input neurons	$28 \times 28 \times 1/32 \times 32 \times 3$
	Number of convolutional layers	2
	Size of convolutional kernel	5×5
	Number of pooling layers	2
	Size of pooling layers	2×2
	Number of fully connected layers	2
	Number of neurons in fully connected layer	120/84
	Activation function	relu
	Number of output neurons	10
LSTM	Number of input neurons	28/32
	Number of hidden layers	4
	Number of neurons in each hidden layer	128
	Number of output neurons	10
	Activation function for hidden layers	relu
	Activation function for the output layer	linear
DESN	Number of input neurons	28/32
	Number of reservoir pools	4
	Reservoir pool size	300
	Number of output neurons	10
	Leakage parameter	0.8
	Spectral radius	0.95
	Reservoir pool activation function	tanh
	Activation function for the output layer	identity

5.2 Dataset and Comparative Methods

In fields such as intelligent surveillance and innovative healthcare in IoT, it is often necessary to classify different images to achieve intelligent management. Therefore, simulations are conducted on datasets related to image classification, and MHSL is compared with other methods. We use 12,000 images for

each dataset as the training dataset and 3,600 images as the testing dataset, evenly distributed among the various clients. Below is a brief introduction to the datasets and comparative methods.

5.2.1 Datasets

- **MNIST Dataset:** A dataset of handwritten digits, consisting of grayscale images of digits from 0 to 9, suitable for quickly verifying the basic functionality of models.
- **FEMNIST Dataset:** A dataset of clothing images, consisting of images of fashion items from 10 categories, more challenging than MNIST and better suited for testing the model's generalization ability.
- **CIFAR-10 Dataset:** A dataset of small color images, consisting of color images from 10 categories, with richer content and colors, requiring more complex models for effective classification.

5.2.2 Comparative Methods

- **Centralized Method (CM):** All data are transmitted to the server for model training. This method serves as the baseline.
- **Federated Learning [10] (FL):** Each client has a local dataset, and FL completes global model training by averaging the local models.
- **Split Learning [5] (SL):** Similar to MHSL, SL places the model part with lower computational requirements on the client, namely the input layer and shallow hidden layers, and the part with higher computational requirements on the server, namely the deep hidden layers and the output layer. SL first trains the client model, uploads the results to the server and then continues training on the server-side model.
- **Hybrid Split and Federated Learning [31] (HSFL):** Each client has a local dataset, and HSFL first uses FL's parallel model training mechanism to train the client model. Then, similar to MHSL, it adopts SL's model splitting structure, with the part requiring higher computation trained on the server.
- **Personalized & Fair Split Learning [32] (PFSL):** Each client has a local dataset, and PFSL combines transfer learning with a lightweight personalized fairness strategy. It completes the training of client models without sharing data or labels with the server.

5.3 Model Performance

Various distributed training methods and different models are trained on three datasets, and their performance is compared in detail, as shown in Figure 3. Across different datasets, it is observed that the performance of the baseline method remains unchanged as the number of nodes varies. This is because the baseline method typically transfers data from all nodes to a centralized location for model training. Thus, its performance remains unaffected by changes in the number of nodes. On the MNIST and FEMNIST datasets, MHSL exhibits smoother performance changes compared to other distributed training methods across different models. HSFL and FL methods follow while SL and PFSL methods perform relatively poorly. This is because MHSL introduces Mixup-based data augmentation, thereby enhancing the

robustness of the model to changes in the number of nodes. In contrast, HSFL and FL adopt parallel client training methods, which partially mitigate the impact of changes in the number of nodes on model training effectiveness. However, SL is significantly affected by changes in the number of nodes due to limitations in its scalability. Transfer learning in PFSL cannot address the scalability issues introduced by model splitting; thus, the performance of PFSL models is significantly affected by changes in the number of nodes. On the CIFAR10 dataset, the overall differences in performance trends among different distributed training methods are insignificant. This is because the complexity of the CIFAR10 dataset is higher compared to the other two datasets; therefore, Mixup cannot generate sufficiently diverse sample variations, and MHSL does not exhibit a significant advantage.

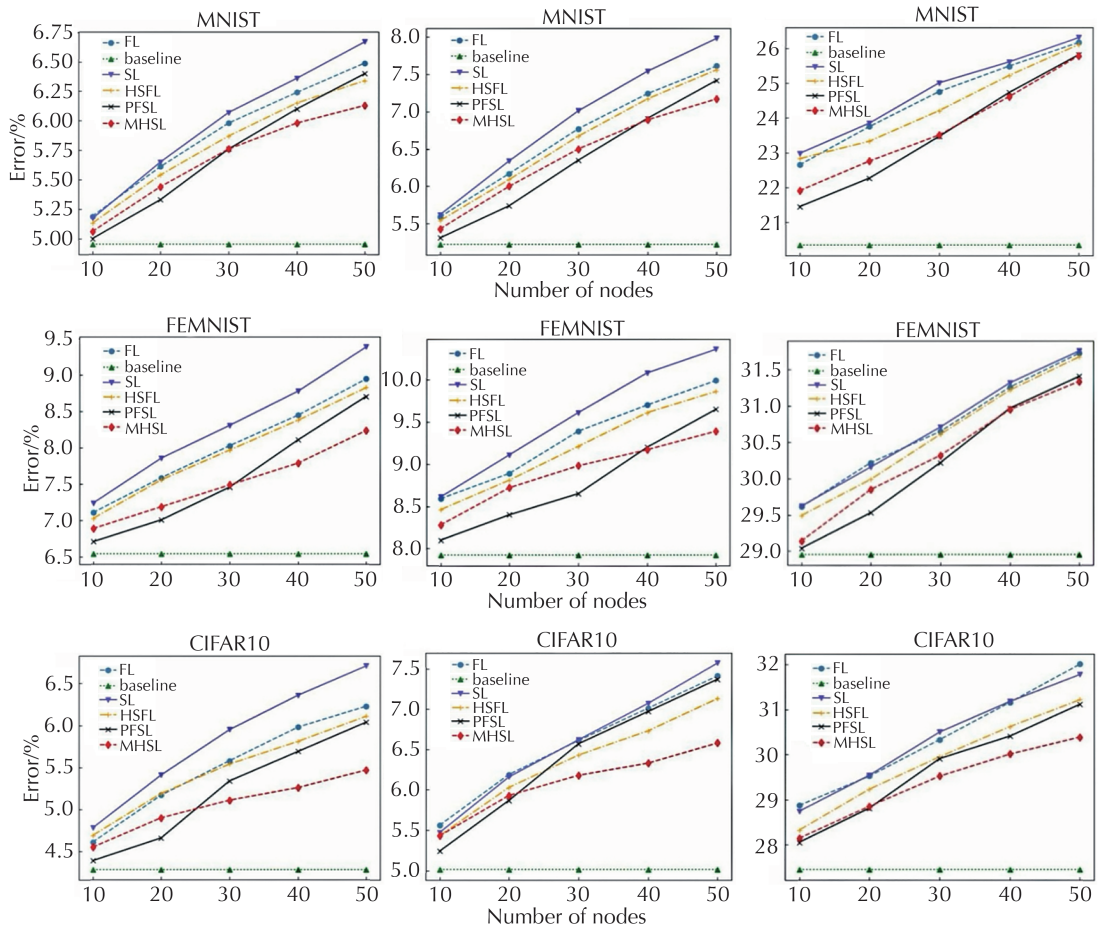


Figure 3. The model performance of different distributed training methods on the MNIST, FEMNIST, and CIFAR10 datasets.

On the other hand, when the number of nodes is small, the model performance of the PFSL method is better than that of MHSL on different datasets. As the number of nodes increases, MHSL maintains optimal model performance on different datasets. This is because HSFL, SL, and FL mainly focus on the split and aggregation methods of model-distributed training. When the number of nodes is small, the transfer learning and fairness strategies on clients in PFSL effectively train personalized client models. However, with the increase in node number, this lightweight personalized strategy gradually loses its effectiveness, leading to a decrease in model performance. MHSL adopts a weight allocation method based on HNs, assigning weight coefficients to each client, ensuring that each client can conduct model training more individually and will not be invalidated by increasing node numbers, thus maintaining overall better model performance.

In conclusion, for different datasets, MHSL generally maintains the optimal performance of various models across different numbers of nodes, demonstrating its overall universality and effectiveness.

5.4 Ablation study

To verify the effectiveness of Mixup in improving the scalability issues of SL, performance comparisons of different models are conducted on each of the three datasets.

Under different datasets, the training results of models with and without Mixup are shown in Figure 4. The formula for calculating the average change rate of model performance is defined as $\frac{1}{n} \sum_{i=1}^n \frac{(error_{i+1} - error_i)}{error_i}$. With the change in the number of nodes, on the MNIST dataset, the average change rates of model performance for CNN, LSTM, and DESN in the MHSL method with Mixup are 4.93%, 4.58%, and 4.73% respectively, while those in the distributed training method without Mixup are 8.06%, 6.30%, and 6.99% respectively. On the FEMNIST dataset, the average change rates of model

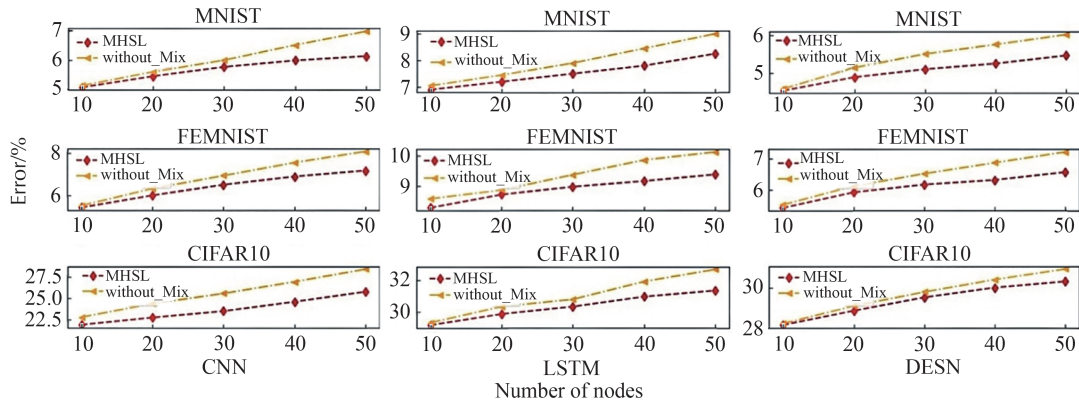


Figure 4. The experimental results of Mixup on CNN, LSTM, and DESN models on the MNIST, FEMNIST, and CIFAR10 datasets.

performance for CNN, LSTM, and DESN in the MHSL method with Mixup are 7.22%, 3.20%, and 4.95% respectively, while those in the distributed training method without Mixup are 9.98%, 4.27%, and 6.96% respectively. On the CIFAR10 dataset, the average change rates of model performance for CNN, LSTM, and DESN in the MHSL method with Mixup are 4.16%, 1.84%, and 1.88% respectively, while those in the distributed training method without Mixup are 5.67%, 2.77%, and 2.34% respectively. Therefore, with the change in the number of nodes, across different datasets, the MHSL method with Mixup exhibits smoother performance changes compared to the distributed training method without Mixup for all three models. This is because Mixup-based data augmentation generates more diverse new samples, presenting smooth transitions in the feature space, which can increase robustness to variations and improve the impact of changes in the number of nodes on model performance.

To validate the effectiveness of HNs in improving the generalization of SL, Ablation study is conducted on the three datasets by varying their degrees of heterogeneity. With the distributed node count fixed at 10, a quantity-based class imbalance method is employed to simulate data heterogeneity scenarios [33], where each client possesses only k different classes of datasets. Initially, k distinct classes of datasets are randomly assigned to each client, and then for each class of dataset, it is evenly distributed among clients that possess that class. Thus, the number of classes for each party remained constant. In the experiments, k is varied from 2 to 4. The performance results across different models are compared.

Under different datasets, the training results of models with and without HNs are shown in Figure 5. The formula for calculating the average model performance error is defined as $\frac{\sum_{i=1}^n error_i}{n}$. With the change in

the heterogeneity of datasets on each node, on the MNIST dataset, the average model performance errors for CNN, LSTM, and DESN in the MHSL method with HNs are 7.67%, 8.43%, and 7.33% respectively, while those in the distributed training method without HNs are 8.70%, 9.67%, and 7.83% respectively. On the FEMNIST dataset, the average model performance errors for CNN, LSTM, and DESN in the MHSL method with HNs are 11.14%, 12.13%, and 9.80% respectively, while those in the distributed training method without HNs are 13.70%, 14.20%, and 12.90% respectively. On the CIFAR10 dataset, the average

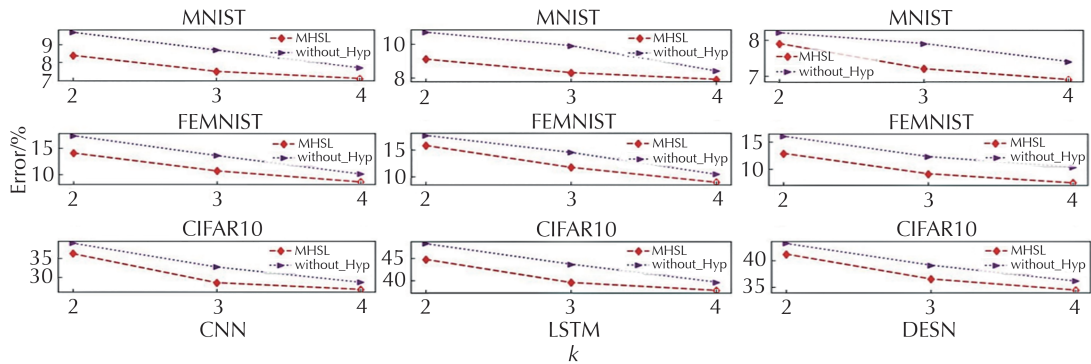


Figure 5. The experimental results of HNs on CNN, LSTM, and DESN models on the MNIST, FEMNIST, and CIFAR10 datasets.

model performance errors for CNN, LSTM, and DESN in the MHSL method with HNs are 30.50%, 40.67%, and 37.37% respectively, while those in the distributed training method without HNs for LSTM and DESN are 33.47%, 43.9%, and 39.47% respectively. Therefore, with the change in dataset heterogeneity, across different datasets, the MHSL method with HNs outperforms the distributed training method without HNs for different models. This is because the weight distribution based on HNs dynamically outputs the weights of input parameters during training execution, enabling the maintenance of personalized models for each client, thus improving the impact of data heterogeneity on model performance.

We further provide a brief sensitivity analysis on key hyperparameters. For Mixup, prior studies [34] typically set λ within the range $\{0.2, 0.5, 0.8\}$, where a smaller value of λ tends to preserve more original features while a larger value of λ encourages stronger blending. For HyperNetworks, architectures with 2–3 fully connected layers are commonly adopted in related work [35], as they offer a reasonable balance between personalization capacity and computational efficiency. Based on these insights, our MHSL framework is expected to remain robust within such ranges, though further empirical evaluation belongs to our future work.

6. CONCLUSION

To effectively utilize the vast amount of data stored in distributed nodes in IoT and achieve efficient distributed training of deep learning models, we propose a model distributed training method based on improved split learning. Firstly, to address the scalability issue of split learning by the limited number of nodes in IoT, Mixup is employed to blend the smashed data generated by client models. Secondly, to tackle the generalization issue of split learning by the heterogeneous data in IoT, HNs are used to determine the weights of each client model. Finally, simulations are conducted on datasets, and the results show that compared to other distributed training methods, the proposed method generally exhibits superior universality and model performance in distributed environments. Moreover, as the number of nodes changes, the proposed method shows an average decrease in model performance variability of 2.37%, 1.95%, and 0.97%, respectively, across the three datasets, demonstrating more stable model performance compared to methods without Mixup. Additionally, as data heterogeneity varies, the proposed method shows an average decrease in model performance error of 0.92%, 2.58%, and 2.97%, respectively, across three datasets compared to methods without HNs, indicating better model performance.

ACKNOWLEDGEMENTS

This work was supported by the National Natural Science Foundation of China (Nos. 61972210, 62102195, 62202237, 61802206, 61803212), the State Key Lab. for Novel Software Technology (No. KFKT2022B04) and the 1311 Talent Program of Nanjing University of Posts and Telecommunications, Natural Science Foundation of Jiangsu Province under Grant No. BK20230351.

REFERENCES

- [1] P. Muralidhara, "IoT applications in cloud computing for smart devices," *International Journal of Computer Science and Technology*, vol. 1, no. 1, pp. 1–41, 2017.
- [2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
- [3] W. Z. Khan, E. Ahmed, S. Hakak, et al., "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, 2019.
- [4] Q. Li, Z. Wen, Z. Wu, et al., "A survey on federated learning systems: Vision, hype and reality for data privacy and protection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 4, pp. 3347–3366, 2021.
- [5] O. Gupta and R. Raskar, "Distributed learning of deep neural network over multiple agents," *Journal of Network and Computer Applications*, vol. 116, pp. 1–8, 2018.
- [6] C. Thapa, M. A. P. Chamikara, and S. A. Camtepe, "Advancements of federated learning towards privacy preservation: from federated learning to split learning," in *Federated Learning Systems: Towards Next-Generation AI*, 2021, pp. 79–109.
- [7] D. J. Han, H. I. Bhatti, J. Lee, et al., "Accelerating federated learning with split learning on locally generated losses," in *ICML Workshop on Federated Learning for User Privacy and Data Confidentiality*, 2021, pp. 1–12.
- [8] M. Duan, D. Liu, X. Chen, et al., "Self-balancing federated learning with global imbalanced data in mobile systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 59–71, 2020.
- [9] C. T. Dinh, N. Tran, and J. Nguyen, "Personalized federated learning with moreau envelopes," in *Advances in Neural Information Processing Systems*, vol. 33, pp. 21394–21405, 2020.
- [10] Y. Huang, L. Chu, Z. Zhou, et al., "Personalized cross-silo federated learning on non-iid data," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 9, pp. 7865–7873, 2021.
- [11] H. Guo, Y. Mao, and R. Zhang, "Mixup as locally linear out-of-manifold regularization," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 1, pp. 3714–3722, 2019.
- [12] R. Zhang, Y. Chen, C. Wu, et al., "Optimizing efficient personalized federated learning with hypernetworks at edge," *IEEE Network*, vol. 37, no. 4, pp. 120–126, 2023.
- [13] J. Su, D. V. Vasconcellos, S. Prasad, et al., "Lightweight classification of IoT malware based on image recognition," in *IEEE Annual Computer Software and Applications Conference*, 2018, pp. 664–669.
- [14] B. Breve, G. Cimino, and V. Deufemia, "Identifying security and privacy violation rules in trigger-action IoT platforms with NLP models," *IEEE Internet of Things Journal*, vol. 10, no. 6, pp. 5607–5622, 2022.
- [15] B. McMahan, E. Moore, D. Ramage, et al., "Communication-efficient learning of deep networks from decentralized data," in *Artificial Intelligence and Statistics*, 2017, pp. 1273–1282.
- [16] Y. Wang, Y. Xu, Q. Shi, et al., "Quantized federated learning under transmission delay and outage constraints," *IEEE Journal on Selected Areas in Communications*, vol. 40, no. 1, pp. 323–341, 2021.
- [17] S. P. Karimireddy, S. Kale, M. Mohri, et al., "Scaffold: Stochastic controlled averaging for federated learning," in *International Conference on Machine Learning*, 2020, pp. 5132–5143.
- [18] X. Wang, Y. Han, C. Wang, et al., "In-edge AI: Intelligentizing mobile edge computing, caching and communication by federated learning," *IEEE Network*, vol. 33, no. 5, pp. 156–165, 2019.
- [19] J. Kang, Z. Xiong, D. Niyato, et al., "Reliable federated learning for mobile networks," *IEEE Wireless Communications*, vol. 27, no. 2, pp. 72–80, 2020.
- [20] P. Vepakomma and R. Raskar, "Split Learning: a resource efficient model and data parallel approach for distributed deep learning," in *Federated Learning: A Comprehensive Overview of Methods and Applications*, Cham: Springer International Publishing, 2022, pp. 439–451.

- [21] E. Samikwa, A. Di Maio, and T. Braun, "Ares: Adaptive resource-aware split learning for internet of things," *Computer Networks*, vol. 218, pp. 1–13, 2022.
- [22] W. Zhou, Z. Qu, Y. Zhao, et al., "An efficient split learning framework for recurrent neural network in mobile edge environment," in *Proceedings of the Conference on Research in Adaptive and Convergent Systems*, 2022, pp. 131–138.
- [23] F. M. Salem, *Recurrent Neural Networks*. Springer International Publishing, 2022.
- [24] C. Thapa, P. C. M. Arachchige, S. Camtepe, et al., "Splitfed: When federated learning meets split learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, pp. 8485–8493, 2022.
- [25] V. Ninkovic, D. Vukobratovic, D. Miskovic, and M. Zennaro, "COMSPPLIT: a communication-aware split learning design for heterogeneous IoT platforms," *IEEE Internet of Things Journal*, vol. 99, pp. 1–12, 2024.
- [26] J. Scott, H. Zakerinia, and C. H. Lampert, "PeFLL: personalized federated learning by learning to learn," *arXiv preprint arXiv:2306.05515*, 2025.
- [27] C. Zhang, H. Li, X. Liu, L. Jiang, and D. Wang, "Hypernetworks for model-heterogeneous personalized federated learning," *arXiv preprint arXiv:2507.22330*, 2025.
- [28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.
- [29] S. Caldas, S. M. K. Duddu, P. Wu, T. Li, J. Konečný, H. B. McMahan, V. Smith, and A. Talwalkar, "LEAF: A benchmark for federated settings," *arXiv preprint arXiv:1812.01097*, 2018.
- [30] A. Krizhevsky, "CIFAR-10 Dataset," 2009. [Online]. Available: <http://www.cs.toronto.edu/kriz/cifar.html>
- [31] X. Liu, Y. Deng, and T. Mahmoodi, "Wireless distributed learning: a new hybrid split and federated learning approach," *IEEE Transactions on Wireless Communications*, vol. 22, no. 4, pp. 2650–2665, 2022.
- [32] M. Wadhwa, G. R. Gupta, A. Sahu, et al., "PFSL: Personalized & Fair Split Learning with Data & Label Privacy for thin clients," in *International Symposium on Cluster, Cloud and Internet Computing*, 2023, pp. 377–390.
- [33] Q. Li, Y. Diao, Q. Chen, et al., "Federated learning on non-iid data silos: An experimental study," in *International Conference on Data Engineering*, 2022, pp. 965–978.
- [34] H. Zhang, M. Cisse, Y. N. Dauphin, et al., "mixup: Beyond empirical risk minimization," *arXiv preprint arXiv:1710.09412*, 2017.
- [35] A. Shamsian, A. Navon, E. Fetaya, et al., "Personalized federated learning using hypernetworks," in *International Conference on Machine Learning*, PMLR, 2021, pp. 9489–9502.