Latency Analysis of Multi-Exit Split Federated Learning via Testbed Implementation

Hyelee Lim, Atif Rizwan, and Minseok Choi
Department of Electronic Engineering, Kyung Hee University, Yongin, South Korea
E-mails: mary@khu.ac.kr, atifrizwan@khu.ac.kr, choims@khu.ac.kr

***Abstract*—Federated Learning (FL) offers a decentralized approach to training machine learning models across distributed devices, addressing data privacy concerns and reducing communication overhead inherent in centralized learning methods. However, as deep neural networks (DNNs) increase in complexity, the memory limitations of client devices pose significant challenges, particularly in real-time applications. To overcome these challenges, Split Learning (SL) has been proposed, which partitions DNN models between client devices and a central server, thereby reducing the burden on clients while maintaining data privacy. This paper extends SL by integrating a multi-exit strategy, allowing for early inference on less complex data samples to further enhance inference speed and efficiency. We validate our approach using a real-world testbed that replicates actual communication environments, contrasting it with existing methods under network conditions. Our results demonstrate that the Multi-Exit Federated Split Learning (ME-FedSL) algorithm not only reduces training and inference times but also improves the feasibility of deploying complex DNN models in federated learning environments, particularly in scenarios with bandwidth constraints.**

***Index Terms*—Federated learning, Split learning, Testbed implementation**

I. Introduction

In traditional centralized machine learning paradigms, large datasets are transferred to a central server where the model training occurs. This approach, while effective, results in substantial communication overhead and raises significant concerns regarding data privacy. To mitigate these challenges, Federated Learning (FL) has emerged as a viable alternative. FL facilitates the decentralized training of models directly on distributed local data, eliminating the need to centralize the data itself. Instead, only the locally trained model parameters are transmitted to a central server for the construction of a global model [1]. However, as deep neural networks (DNNs) grow in complexity with deeper architectures, client devices—which generally possess limited memory and computational resources compared to central servers—face considerable delays during the training or inference of these extensive models. This performance gap underscores the critical need for efficient model compression and optimization techniques, particularly for real-time applications on client devices where resource constraints are a key consideration.

To address the challenges of training DNNs on resource-constrained client devices, Split Learning (SL) has emerged as a promising solution. SL strategically divides a DNN model into two separate model blocks and store them in the client and server separately. In this method, the client device trains the model up to the split layer and then sends the resulting activations to the server, which continues the training for the remaining layers [2]. This approach significantly alleviates the computational load on client devices by limiting their responsibility to only a portion of the model, thereby reducing the overall resource demand compared to training the entire DNN. Accordingly, SL addresses the limitations of FL on the client side by reducing the computational and resource demands, ultimately enhancing both training and inference speeds on client devices with limited computational resources.

Neural networks, characterized by their hierarchical feature extraction capabilities, have driven the development of increasingly deep and complex models to achieve higher performance. However, the addition of layers in a deep neural network, while beneficial for accuracy, also results in increased latency and computational overhead during inference. These challenges are particularly problematic in real-time applications and on devices with limited computational resources. To mitigate these issues, [3] introduced a novel architecture that incorporates side branches into the neural network, facilitating early inference. This design allows for test samples that can be classified with high confidence at earlier layers to exit the network without processing through all subsequent layers. In result, more complex samples are forwarded to deeper layers, while simpler ones are efficiently processed at earlier stages. This early exit mechanism reduces the computational load on the deeper layers, thereby enhancing inference speed and making the model more suitable for real-time and resource-constrained environments.

The authors of [2] argued that training time could be improved, presenting results based on software simulations. However, there is a scarcity of studies that have validated these claims using actual testbeds. Although simulations can consider limited resources and communication environments, they do not fully replicate real-world communication conditions and computational resources. To overcome these limitations, this paper establishes a testbed using NVIDIA Jetson Nano boards to reflect real communication environments and resource constraints, aiming to validate the performance of the proposed algorithm under realistic conditions. Additionally, we integrate a multi-exit neural network architecture [3] into the ME-SplitFed framework [2] to improve the inference latency performance. Through this empirical validation, we demonstrate that the proposed method can indeed improve inference speed in practical environments.

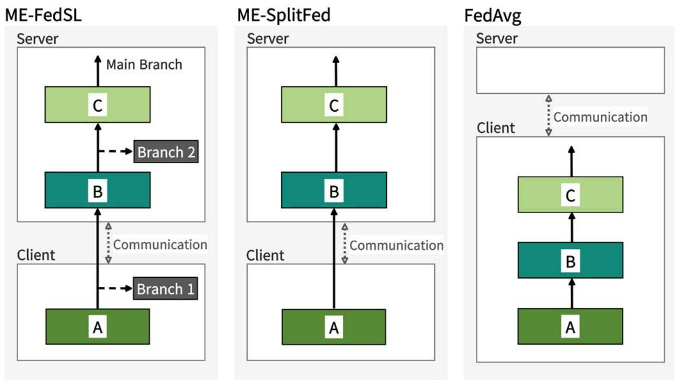

Fig. 1: Comparison of Algorithms

II. Multi-Exit Split Federated Learning Algorithm

An early FL technique, FedAvg, involves multiple clients independently training their local models on their respective local datasets. The central server then aggregates the model parameters from each client by computing a weighted average to create a global model [1]. Specifically, the objective is to learn a global model $\mathbf{w}$ that minimizes the global loss function:

$$F(\mathbf{w}) = \sum_{k=1}^{K} \frac{n_k}{n} F_k(\mathbf{w}), \tag{1}$$

where $F_k(\mathbf{w})$ represents the local loss function for client k, n_k is the number of data samples at client k, and $n = \sum_{k=1}^{K} n_k$ is the total number of data samples across all clients.

The resulting global model $\mathbf{w}$ is then distributed to all clients, who use it as the basis for further local training. The local objective function for each client k is defined as:

$$F_k(\mathbf{w}) = \frac{1}{n_k} \sum_{i=1}^{n_k} \ell(\mathbf{w}, \mathbf{x}_i, y_i), \tag{2}$$

where $\ell(\mathbf{w}, \mathbf{x}_i, y_i)$ represents the loss function, and $\mathbf{x}_i$ and y_i are the input data and the corresponding label for the i-th data point in client k's dataset, respectively. Through iterative rounds of local training and global aggregation, FedAvg enables distributed learning on individual devices while ensuring that local data remains private. This process culminates in the convergence of a global model $\mathbf{w}$ that optimizes the overall loss function across all participating clients [1].

We first split the given deep learning model $\mathbf{w}$ into the client-side model block $\mathbf{w}_C$ and server-side model block $\mathbf{w}_S$. For client k's model, we denote client-side and server-side model blocks by $\mathbf{w}_{C,k}$ and $\mathbf{w}_{S,k}$, respectively. Different from the existing split FL approach in [2], we allow early prediction by adding auxiliary classifiers to the intermediate layers of the given deep learning model, as [3] and [4] did. In particular, the auxiliary classifier ϕ_1 for the first early exit is added to the cut layer of $\mathbf{w}_C$, and the auxiliary classifier ϕ_2 for the second early exit is added in the middle of $\mathbf{w}_S$. Without loss of generality, we assume $\|\mathbf{w}_C\|_0 \ll \|\mathbf{w}_S\|_0$ to save the storage space of the client device.

At the beginning of the training phase, the parameter server broadcasts an initial model $\mathbf{w}^0$ to all clients, i.e., $\mathbf{w}_k^{0,0} = \mathbf{w}^0$, where $\mathbf{w}_k^{t,e}$ is the local model of client k at the e-th local epoch in the t-th global round. Every client independently performs the local training process based on the mini-batch stochastic gradient descent. Since we have three different exits, the client should define a new multi-exit loss function including all the loss functions from different exits, as given by

$$F_k(\mathbf{w}_k) = \gamma_1 f_k(\mathbf{w}_{C,k}, \phi_1) + \gamma_2 f_k(\mathbf{w}_k, \phi_2) + (1-\gamma_1-\gamma_2) f_k(\mathbf{w}_k), \tag{3}$$

where $f_k(\mathbf{w}_{C,k}, \phi_1)$ and $f_k(\mathbf{w}, \phi_2)$ are the loss functions of two early exits, and $f_k(\mathbf{w})$ is the loss function obtained at the final layer of $\mathbf{w}$. We can control the performances of early predictions by controlling γ_1 and γ_2, but we assume $\gamma_1 = \gamma_2 = 1/3$ in this paper to focus solely on the impact of model split and multi-exit architecture on the training and inference latency. Based on the multi-exit objective function in 3, client k updates its local model as given by

$$\mathbf{w}_k^{t,e+1} \leftarrow \mathbf{w}_k^{t,e} - \eta \nabla F_k(\mathbf{w}_k^{t,e}), \tag{4}$$

where η is the learning rate. Here, to calculate $F_k(\mathbf{w}_k^{t,e})$, we need to obtain the loss functions of the server-side model block. Therefore, client k uploads the activation values of the cut layer $\varphi_{C,k}^{t,e}$ of $\mathbf{w}_{C,k}^{t,e}$ with respect to the input x_i, denoted by $a_{C,k}(\varphi_{C,k}^{t,e}; x_i)$, and its label y_i to the server every local epoch of the local training process. The client use $a_{C,k}(\varphi_{C,k}^{t,e}; x_i)$ to update the client-side model branch ϕ_1 and transmits the $a_{C,k}(\varphi_{C,k}^{t,e}; x_i)$ and loss $\ell^{\phi_1} = f_k(\mathbf{w}_{C,k}, \phi_1)$ to the server. Then, the server continues the forward propagation process with $a_{C,k}(\varphi_k^{t,e}; x_i)$ for its model block, $\mathbf{w}_{S,k}^{t,e}$ to calculate and computes the activations $a_{S,k}(\varphi_{S,k}^{t,e}; x_i)$ on server-side cut layer $\varphi_{S,k}^{t,e}$. These activations are then used to compute $f_k(\mathbf{w}_k, \phi_2)$ and $f_k(\mathbf{w}_k^{t,e})$. Then, the server performs the back-propagation process and delivers the gradient of the cut layer, denoted by $\mathbf{g}_k(\varphi_{C,k}^{t,e})$, and loss function values, $f_k(\mathbf{w}_k^{t,e}, \phi_2)$ and $f_k(\mathbf{w}_k^{t,e})$ to client k. This process iterates for E local epochs, and we have $\mathbf{w}_k^{t,E} = [\mathbf{w}_{C,k}^{t,E}, \mathbf{w}_{S,k}^{t,E}]$.

After E local epochs, the model aggregation step begins. all clients upload their updated client-side model blocks, $\mathbf{w}_{C,k}^{t,E}$, to the server, and the server takes the weighted average of them, as given by

$$\bar{\mathbf{w}}_C^t = \sum_{k=1}^{K} \frac{n_k}{n} \mathbf{w}_{C,k}^{t,E}. \tag{5}$$

Afterwards, the aggregated client-side model block $\bar{\mathbf{w}}_C^t$ is broadcasted to all clients, and clients set $\mathbf{w}_{C,k}^{t+1,0} \leftarrow \bar{\mathbf{w}}_C^t$ as an initial model for the next global round. Here, note that we do not aggregate the server-side model blocks as the authors of [2] did. Algorithm 1 summarizes the ME-FedSL approach.

Algorithm 1 ME-FedSL

```
Input: Dataset $D_k$ for client $k$, Initial Model Weights $\mathbf{w}_k^{0,0}$, Global Rounds $T$, Local Epochs $E$, Batch Size $B$
Output: $\mathbf{w}_k^T = [\mathbf{w}_{C,k}^T, \mathbf{w}_{S,k}^T]$
 1: Register client on server (using client ID);
 2: for global round $t \in \{1, \dots, T\}$ do
 3:     for each client $k \in \{1, \dots, K\}$ in parallel do
 4:         for local epoch $e \in \{1, \dots, E\}$ do
 5:             for each mini-batch $b \in \{1, \dots, B\}$ do
 6:                 Extract mini-batch $D_k^b \subset D_k$
 7:                 Client-side
 8:                 Compute intermediate output: $a_{C,k}(\varphi_{C,k}^{t,e}; D_k^b)$ and $\ell^{\phi_1} = f_k(\mathbf{w}_{C,k}^{t,e}, \phi_1^{t,e}; D_k^b)$
 9:                 Send $a_{C,k}(\varphi_{C,k}^{t,e}; D_k^b), \ell^{\phi_1}$ and $y$ to server
10:                 Server-side
11:                 Update server model $\mathbf{w}_{S,k}^{t,e}$ using $a_{C,k}(\varphi_{C,k}^{t,e}; D_k^b)$
12:                 Compute $a_{S,k}(\varphi_{S,k}^{t,e}; D_k^b)$
13:                 $\ell^{\phi_2} \leftarrow f_k(\mathbf{w}_{S,k}^{t,e}, \phi_2^{t,e}; a_{C,k}(\varphi_{C,k}^{t,e}; D_k^b), y)$
14:                 $\ell^{\mathbf{w}} \leftarrow f_k(\mathbf{w}_{S,k}^{t,e}; a_{C,k}^{t,e}, y)$
15:                 Compute $\ell_{\text{avg}}$ using Eq. (3)
16:                 Server-side back propagation and compute gradients $\mathbf{g}_k(\varphi_{C,k}^{t,e})$
17:                 Send gradients $\mathbf{g}_k(\varphi_{C,k}^{t,e})$ and $\ell_{\text{avg}}$ to client
18:                 Client-side
19:                 Update client model using received gradients $\mathbf{g}_k(\varphi_{C,k}^{t,e})$
20:                 $\mathbf{w}_{C,k}^{t,e+1} \leftarrow \mathbf{w}_{C,k}^{t,e} - \eta \nabla \mathbf{g}_k(\varphi_{C,k}^{t,e})$
21:             end for
22:         end for
23:         Client: Send updated local weights $\mathbf{w}_{C,k}^{t,E}$ to server
24:     end for
25:     Server: Aggregate client-side models using Eq. (5)
26:     Client: Receive global model weights $\bar{\mathbf{w}}_C^t$ from server
27:     Client: Update client model: $\mathbf{w}_{C,k}^{t,E} \leftarrow \bar{\mathbf{w}}_{C,k}^t$
28: end for
```

III. Implementation and Experimental Results

A. Implementation Setup

For the testbed experiments, five Nvidia Jetson Nano devices were used as clients, and the server's software environment included Windows 11, Python v3.10.9, PyTorch v2.0.1, CUDA v11.7, and CuDNN v8.5.0. The hardware configuration for the server consisted of a desktop equipped with a GTX 1660 SUPER GPU and an Intel i5-11660k CPU. The Nvidia Jetson Nano devices operated on Ubuntu 18.04, with Python v3.6.9, PyTorch v1.8.0, CUDA v10.2.89, and CuDNN v8.0.0. The hardware specifications of the Jetson Nano included a 128-core NVIDIA Maxwell GPU and a Quad-Core ARM A57 CPU. The communication environment was wireless, utilizing an ipTIME A2004MU router that supports both 5GHz and 2.4GHz bandwidths. The wireless network cards inserted into the Jetson Nano devices were Intel 8265NGW, supporting a maximum speed of 867Mbps on the 5GHz band. The CIFAR-10 dataset was used, with each client following an IID distribution without overlap. The network model employed for training was ResNet-110, a simplified version of the original ResNet model with reduced ResBlock feature sizes. The training hyperparameters were set as follows: 20 global rounds, 3 local epochs, a batch size of 32, and a learning rate of 0.001. The Adam optimizer (betas of 0.9 and 0.999) and the cross-entropy loss function were used. Inference was performed using the CIFAR-10 test dataset, consisting of 10,000 samples, with a batch size of 32. This setup provided a standardized environment for evaluating the model's inference performance, allowing for the measurement of both prediction accuracy and processing speed.

The server and client exchange data through TCP socket communication. The server waits for connection requests from the client, and the client initiates a connection to the server. During this process, both the server and the client create sockets. The server binds its socket to a specific IP address and port number, waiting for connection requests from clients. The client sends a connection request using the server's IP address and port, and the server accepts this request. In our experiment, the server can create a single socket, so clients have to wait for uploading their model parameters while the other one is communicating with the server. In addition, before data transmission begins, a connection must be established through a 3-way handshake, which involves multiple signal exchanges between both sides. Similarly, when terminating the connection, a 4-way handshake is required. As a result, communication overhead can arise during both the connection setup and termination phases; however, their latency is not really significant compared to the computation delay at the client device and communication latency.

We compare the ME-FedSL with the existing approaches of ME-SplitFed [2] and FedAvg [1] in our testbed implementations using NVIDIA Jetson Nano boards. Note that the clients of FedAvg store and compute the full model and ME-SplitFed allows clients to exchange the activation and gradient values with the server in the training phase. On the other hand, in the inference stage, ME-SplitFed always needs to compute the full model to get the final prediction result. Also, we summarize the simulation environments with different bandwidth and computing resource availability.

TABLE I: Evaluation of network bandwidth performance.

Bandwidth (Mbits/sec)					
Maximum			Minimum		
Server	Jetson Nano		Server	Jetson Nano	
receiver	receiver	sender	receiver	receiver	sender
184	185	186	66.1	66.4	67.4

1) Normal Setting: For the normal setting, a wireless 802.11ac (5G) network, which supports theoretical maximum speeds up to 650 Mbps (TX Bitrate: 650.0 MBit/s, VHT-MCS 7, 80MHz, short GI, VHT-NSS 2), along with the Jetson Nano boards' GPU clock set to 921 MHz and the CPU clock set to

1.5 GHz on all quad cores for maximum performance, are assumed.

2) Bandwidth Performance Degradation: A wireless 802.11n (2.4G) network, with a maximum achievable communication rate of 270 Mbps (TX Bitrate: 270.0 MBit/s, MCS 14, 40MHz, short GI), along with the Jetson Nano boards' GPU clock set to 921 MHz and the CPU clock set to 1.5 GHz on all quad cores for maximum performance, are assumed.

B. Experimental Results

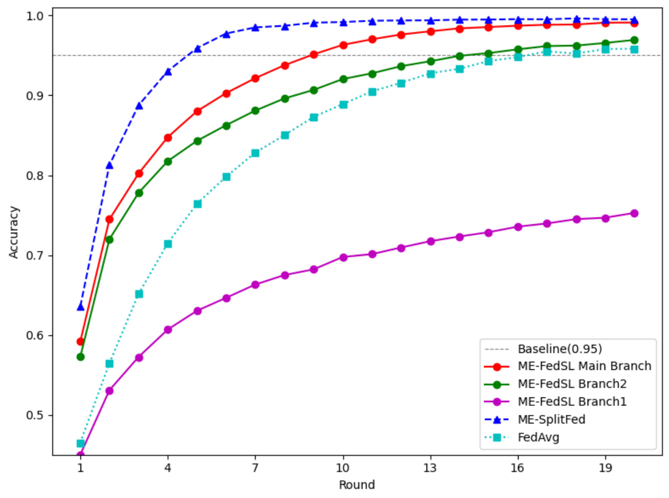

Fig. 2: Normal Setting Experiment; Overall Algorithm Results

1) Baseline Experiment: This experiment was conducted under maximum bandwidth conditions, with the Jetson Nano devices operating at their maximum GPU clock speed. During the experiment, the average communication speed between the server and the Jetson Nano devices was measured at 185 Mbits/sec, as shown in Table 1, which presents the results of the average bandwidth speed experiment between the desktop and Jetson Nano. The training results visualized in Figure 2 illustrate the number of global rounds required to reach the target accuracy. The time required for each algorithm to achieve the target accuracy of 95% was calculated as follows, as shown in Table 2: ME-FedSL 20,691.71 seconds, ME-SplitFed 8,757.20 seconds, and FedAVG 55,976.03 seconds. Among these, FedAVG, which requires the most computation, had the longest round time, averaging 3,292.7081 seconds per round. ME-FedSL and ME-SplitFed are algorithms that split the model between the client and server for training. Compared to FedAVG, they require less communication and computation per round, resulting in faster training times. Notably, ME-FedSL employs a branch structure that allows for early processing of some data on the client side, leading to additional performance improvements over ME-SplitFed. Test results showed that ME-FedSL, utilizing the branch structure, processed the entire test dataset approximately 0.1023 seconds faster than ME-SplitFed(Table 3). These findings demonstrate that the introduction of a branch structure in ME-FedSL can significantly enhance training and inference performance. This is particularly beneficial in applications where real-time performance is crucial, as the branch structure in ME-FedSL enables quicker predictions, thus improving overall performance. Moreover, ME-FedSL shows potential for maintaining high performance in resource-constrained environments, proving to be more efficient than FedAVG in scenarios with sufficient communication bandwidth. FedAVG's slower performance is primarily due to its approach of training the entire model and uploading model weights to the server at the end of each round, leading to higher computation and communication costs. In contrast, ME-FedSL and ME-SplitFed algorithms split the model between the client and server, distributing the computational load and reducing the frequency and volume of data transmissions, resulting in faster training times. As shown in the test speed comparison in Table 3, ME-FedSL completed tasks faster than ME-SplitFed, likely due to the branch mechanism that allows test samples to be processed more quickly. These results suggest that split learning-based algorithms can significantly improve real-time performance and that ME-FedSL may be more advantageous than FedAVG in environments with high computational and communication costs.

2) Bandwidth Variation Experiment: This experiment was conducted under minimum bandwidth conditions, based on the Baseline experiment. During this experiment, the average communication speed between the server and Jetson Nano devices was measured at 66 Mbit/s, approximately 2.8 times lower than the maximum bandwidth, as shown in Table 1. The time required for each algorithm to reach the target accuracy of 95% was calculated as follows, as shown in Table 2 and Figure 2: ME-FedSL 36,265.27 seconds, ME-SplitFed 17,760.90 seconds, and FedAVG 55,637.99 seconds. When comparing the time per round with the Baseline, ME-FedSL and ME-SplitFed exhibited a much larger increase in time per round compared to FedAVG, as ME-FedSL and ME-SplitFed require communication at each iteration during training, whereas FedAVG only transmits model weights. In bandwidth-constrained environments, the communication overhead for ME-FedSL and ME-SplitFed was significantly higher than that for FedAVG. This is because split learning algorithms involve more frequent and larger data exchanges between the client and server, leading to a sharp decrease in training speed as the bandwidth narrows. On the other hand, FedAVG, with its lower communication frequency and only model weight transmission, maintained relatively stable training times despite reduced bandwidth. When the bandwidth was reduced, the inference process for ME-FedSL exhibited an additional delay of 0.1066 seconds, while ME-SplitFed showed a delay of 0.1343 seconds. Despite this, ME-FedSL, which utilizes a branch structure, still maintained a faster test speed by approximately 0.1299 seconds compared to ME-SplitFed ,as shown in Table 3. This suggests that ME-FedSL can deliver relatively higher performance even in environments with limited bandwidth, thanks to its efficient computation distribution and rapid prediction capabilities. The structural advantage of ME-FedSL's branch mechanism particularly con-

TABLE II: The latency of performing a global round in the training phase for the baselines [sec]

Setting	**Normal**			**Limited bandwidth**		
Algorithms	ME-FedSL	ME-SplitFed	FedAvg	ME-FedSL	ME-SplitFed	FedAvg
min	2496.8066	1692.2857	3276.9476	4424.3317	3449.1432	3244.9869
max	2681.5471	1884.4061	3306.4212	4762.1945	3597.1446	3319.4396
avg	2586.4464	1751.4414	3292.7081	4533.1592	3552.1811	3272.8231

TABLE III: The latency of predicting the given test set in the inference phase for the baselines [sec]

Setting	**Normal**		**Limited bandwidth**	
Algorithms	ME-FedSL	ME-SplitFed	ME-FedSL	ME-SplitFed
min	0.020989	0.019996	0.020508	0.020998
max	0.059512	0.057134	0.052027	0.053602
avg	0.026972	0.027299	0.027314	0.027730
total	8.415221	8.517521	8.521905	8.651830

tributed to maintaining the model's inference capabilities and real-time responsiveness effectively, even under degraded communication conditions. These findings suggest that FedAVG may be more advantageous in environments with poor communication conditions, and that split learning algorithms, like ME-FedSL, require optimization based on the communication environment. Furthermore, additional experiments are necessary to evaluate the performance of these algorithms under various bandwidth conditions to better understand the impact of communication environments on learning performance. By doing so, we can identify communication optimization strategies to maximize the efficiency of split learning algorithms, thereby enhancing their applicability in real-world scenarios.

IV. Conclusion

This paper compares and analyzes three algorithms—ME-FedSL, ME-SplitFed, and FedAVG—to enable effective learning in environments with limited communication speed. The experimental results demonstrate that ME-FedSL and ME-SplitFed improve real-time performance due to their split structure and the use of branches for rapid predictions. However, when bandwidth was reduced, both ME-FedSL and ME-SplitFed experienced a significant increase in communication overhead, leading to slower training times compared to FedAVG. Despite its higher computational cost, FedAVG outperformed in environments with limited communication speed due to its lower communication frequency and model weight transmission, which prevented significant drops in learning speed even under poor communication conditions. These findings highlight the importance of considering both computational resources and communication constraints when selecting an algorithm, suggesting that the optimal approach must be tailored to the specific communication environment. Additionally, the study suggests that while ME-FedSL's branch mechanism provides structural advantages in real-time responsiveness, further optimization is necessary to enhance its adaptability and efficiency in bandwidth-constrained scenarios. By refining these split learning algorithms, particularly ME-FedSL, their practical applicability in diverse communication environments can be significantly improved, ensuring more efficient learning under various conditions.

Acknowledgment

This work was supported in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-02201, Federated Learning for Privacy-Preserving Video Caching Networks), in part by the National Research Foundation of Korea(NRF) grant funded by the Korea govenment(MSIT) (NRF-2022R1C1C1010766), and in part by NRF grant funded by MSIT (No. 2022R1A4A3033401).

References

[1] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. Agüera y Arcas, "Communication-efficient learning of deep networks from decentralized data," arXiv, 2023. [Online]. Available: https://arxiv.org/abs/1602.05629

[2] C. Thapa, M. A. P. Chamikara, S. Camtepe, and L. Sun, "SplitFed: When federated learning meets split learning," arXiv, 2022. [Online]. Available: https://arxiv.org/abs/2004.12088

[3] S. Teerapittayanon, B. McDanel, and H. T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," arXiv, 2017. [Online]. Available: https://arxiv.org/abs/1709.01686

[4] D.-J. Han, D.-Y. Kim, M. Choi, C. G. Brinton, and J. Moon, "SplitGP: Achieving both generalization and personalization in federated learning," arXiv, 2023. [Online]. Available: https://arxiv.org/abs/2212.08343