Estimating the Training Time in Single- and Multi-Hop Split Federated Learning

Joana Tirana
joana.tirana@ucdconnect.ie
University College Dublin
Ireland

Spyros Lalis
lalis@uth.gr
University of Thessaly
Greece

Dimitris Chatzopoulos
dimitris.chatzopoulos@ucd.ie
University College Dublin
Ireland

Abstract

Split Federated Learning (SFL) is an upcoming and promising approach that balances the two main goals of distributed training, i.e., (i) the data remains at the data owners, and (ii) even devices with resource limitations can participate in the training. This is achieved by splitting the model into multiple parts and offloading them to designated compute nodes. Recent findings show that the number of compute nodes (*hops*) plays a significant role in the training delay. However, determining the ideal number of hops is not an easy task. Therefore, in this work, we propose a mathematical model that estimates the training delay of single- and multi-hop SFL. This tool not only helps in searching the optimal number of hops before the real deployment happens but also can be used as a lightweight evaluation tool in future research works in SFL. Our numerical evaluations show that the model can make correct estimations with an error smaller than 3.86%. Finally, we have constructed a lightweight optimization problem that finds the optimal cut layers (split points) and model part assignment to minimize training delay.

CCS Concepts

• **Computing methodologies → Distributed artificial intelligence**; **Model verification and validation**.

Keywords

Split Federated Learning, Distributed Learning Modeling

ACM Reference Format:
Joana Tirana, Spyros Lalis, and Dimitris Chatzopoulos. 2025. Estimating the Training Time in Single- and Multi-Hop Split Federated Learning. In *The 8th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3721888.3722096

1 Introduction

Split Federated Learning (SFL) [12], enables the training of Machine Learning (ML) models in a distributed manner while keeping the data decentralized. In a nutshell, SFL is the merge of two training protocols, the Split Learning [15] and Federated Learning (FL) [8]. Specifically, the *data owners*, typically resource-constrained devices, collaborate with designated *compute nodes* to train the ML model for multiple epochs until it converges. The data owners have access to the training data but do not wish to share it. However, their resource limitations prevent them from performing fully on-device training, as in FL. Consequently, the model is split into one or more parts, which are offloaded to more powerful *compute nodes*.

SFL is a recently proposed and upcoming approach and has multiple variations. In this work, we consider SplitFedV1 [12], which is also found as Parallel SL [14] in the literature. This version of SFL is closer to FL, as each data owner trains a different instance of the model, that get aggregated (e.g., using `FedAvg` [8]) at the end of each training round. Whereas, in the other SFL versions (e.g., SplitFedV2 [12], SplitNN [15]) the data owners have at least one shared model part that gets updated sequentially. Further, we adopt the USplit [15] setup, which ensures that the data owners always manage the input and output layers. This setup is considered more privacy-preserving, as the data owners do not share any information regarding the data inputs or labels.

A recent work in SFL [13], studies the role of the compute nodes, by evaluating the batch delay while adding more *hops*, i.e., adding more compute nodes with each of them being in charge of a different model part. An interesting insight from this analysis was that such an arrangement provides a different type of parallelism, the so-called *Pipelined Parallelism* (PP). The analysis in [13] showed that the batch delay is tightly coupled with the number of hops. However, the optimal number of hops (as a measurement of training acceleration) is sensitive to several parameters, e.g., system characteristics such as the computing capacity of the devices and the network, but also training characteristics such as the batch-size, and the complexity of the model. Moreover, the PP adds additional complexity to the process; making it even more difficult to estimate the expected training delay. As a result, deciding which number of hops is optimal for each system and ML case is not an easy task.

Thus, in this work, we propose a mathematical model for estimating the training time in SFL. In detail, given the system and training characteristics, the model predicts the expected training delay. Researchers can use the model to explore and optimize SFL setups without the cost of deployment. Also, the proposed tool serves as an evaluation framework for the SFL research community, facilitating future studies. Since distributed learning protocols require numerous devices and substantial resources, leading to significant costs, a mathematical model can greatly reduce these expenses and act as a pre-deployment assessment tool. Finally, we extend the mathematical model into a *lightweight* optimization problem that determines (i) the optimal cut layers (split points) and (ii) assignment of the

Acknowledgments: This work has been supported by EU Horizon project no 101092912 (MLSysOps).

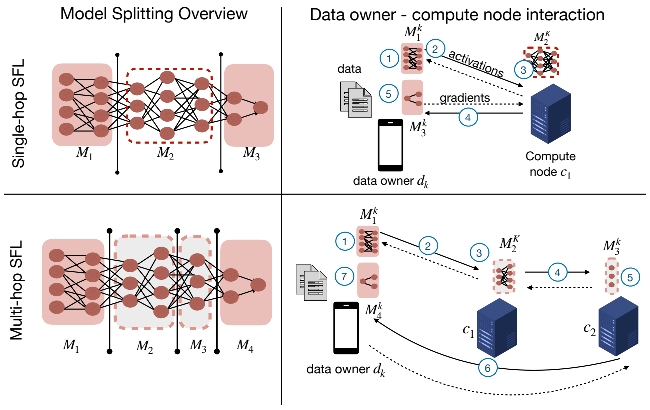


Figure 1: Model splitting (left) and workflow (right) for single- (top) and multi- (bottom) SFL. It enumerates only the steps of the fwd() for simplicity.

model parts into the compute nodes, that minimize training delay. This proves that the proposed model can be easily extended and used as an optimization tool for several system aspects.

In summary, the contributions of the paper are the following,

1– Complexity analysis. Construct an analytical model for estimating the training time of SFL with one or multiple hops.

2 – Multi-hop optimization. Extend the model to optimize the splitting selection and model part allocation. To our knowledge, this is the first work to address optimization for multi-hop SFL.

3 – Validation. Perform numerical evaluations of the proposed methods using collected data from our testbed. The results show that our model accurately estimates training time across all experiments, with an error rate of less than 3.86%.

2 System Model

Model Splitting. Let M be the ML model that is trained using SFL. Also, let mem_M represent the required memory for hosting the model on a node. As SFL requires, the model is split into P parts, e.g., $M_p, 1 \leq p \leq P$, with each part consisting of one or more consecutive layers of the model. Further, we consider a set of data owner nodes $d_k, k \in \mathcal{K}$, with $|\mathcal{K}| = K$ who wish to collaboratively train the model M, using their local data. For simplicity and w.l.o.g., we assume that each data owner has the same number of batches B. Moreover, given that we adopt the USplit setup [15], each data owner d_k locally maintains a unique version of the first and the last model parts of the model, i.e., M_1^k, M_P^k. Consecutively, the data owners allocate $mem_1 + mem_P < mem_M$ memory. Note that, the number of hops, N, is defined by the number of model parts ($P = N + 2$, since M_1 and M_P are assigned by default at the data owners). For example, (left) Fig. 1 shows the model parts with one hop (two cut layers, i.e., 3 model parts), and two hops (three cut layers, i.e., 4 model parts).

Extending this, N compute nodes, denoted as $c_i, i \in \mathcal{N}$, will contribute to the training, by managing the intermediate model parts; one different version per data owner. For that, we use n_p^k to represent the node to which M_p is assigned for the data owner d_k. While $mem_p < mem_M$ is the memory to host M_p, and $procT_p(n)$ is the processing time required to train M_p on node n. Yet, due to USplit, $n_1^k = n_P^k = d_k$, e.g., the first and last parts are always assigned to the respective data owner. While, each of the intermediate parts is assigned to a different compute node, thus $n_p^k \neq n_{p'}^k, 2 \leq p \neq p' \leq P-1$, and the same intermediate part is hosted on the same compute node for all data owners, $n_p = n_p^k = n_p^{k'}, 2 \leq p \leq P-1$.

Batch update. As is shown in the right Fig. 1, data owners communicate with compute nodes over wired or wireless links while compute nodes communicate with each other typically over a fast wired network. W.l.o.g., let $bw_{n_i,n_j} = bw_{n_j,n_i}$[1] denote the bandwidth of the (symmetrical) link between nodes n_i and n_j (data owners or compute nodes). Focusing on the top part of Fig. 1 that shows one *batch update* between one data owner and one compute node (single hop), we see that the data owner d_k initiates the forward-propagation operation (e.g., `fwd()`), by inputting the batch of data into M_1^k (step 1). Then, it sends the intermediate activation of the first cut layer to the compute node (step 2). Next, the compute node applies `fwd()` at M_2^k and sends the activation of the second cut layer to the data owner (steps 3 and 4). This phase is completed when the data owner, applies `fwd()` at the final model part, M_3 to compute the predicted label (step 5). Then, the backward-propagation starts, and the data owner computes the loss (i.e., the difference between the predicted and real label). The loss is back-propagated (e.g., `back()`); following the reverse path. During this operation, the two entities compute the gradients and update the model parts. Similarly, the bottom part of Fig. 1 shows the steps for two hops, i.e., another compute node is added.

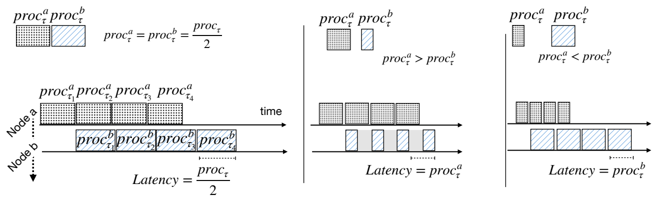
3 Estimating Training Time in SFL

Considering the system model described, we construct a general model for approximating time in SFL. We start our analysis with multi-hop SFL and then, show how the model can be used for one-hop SFL. In multi-hop SFL, multiple compute nodes manage different parts of the model while, data owners update separate instances of the model, enabling parallel updates. This setup allows each compute node to process a different data owner simultaneously. For example, consider the setup in the bottom Fig. 1 with more data owners. In that case, when compute node c_1 is applying `fwd()` on M_2^k, c_2, can be processing $M_3^{k'}$, for $k \neq k'$. This setup resembles a pipeline architecture [13]. Yet, the training for each batch of d_k is performed through a *bidirectional* pipeline. In detail, during the forward-propagation phase, n_p performs the `fwd()` computation for M_p^k and sends the cut layer's activations to n_{p+1} which then starts the `fwd()` for M_{p+1}^k. Conversely, during the back-propagation phase, n_{p+1} computes and sends the respective gradients from M_{p+1}^k to n_p which then starts the `back()` for M_p^k.

Fig. 2 presents some (toy) examples of how the *pipeline delay* is adjusted in different scenarios. It shows that the training tasks are split into two or three parts and are assigned to a corresponding number of nodes. The latency can then be reduced by up to half and, respectively, one-third of the original processing time, provided the task is split evenly among the nodes (left scenarios). Otherwise, the latency is determined by the slowest node (middle and right scenarios). To this end, we make the following observation,

<u>**Observation 1:**</u> *The pipeline's latency (completion time for all model parts in the pipeline) in each direction is defined by the slowest node.*

[1]This assumption is only made to simplify the notation. When evaluating the model we use the real bandwidth value for all network directions. This is a common practice [14].


(a) Each task τ_x is split in two parts assigned to nodes a and b.

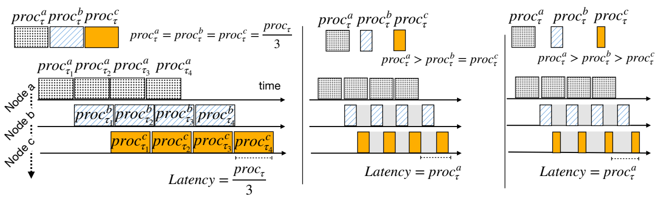

(b) Each task τ_x is split in three parts assigned to nodes a, b and c.

Figure 2: Pipeline latency for different task splitting scenarios. Four training tasks $\tau_1, \tau_2, \tau_3, \tau_4$ are split into smaller parts, assigned to (a) two or (b) three nodes in a pipeline (vertical direction). The boxes represent the time needed by the nodes to process their parts (horizontal direction). The grey areas denote the waiting times for the completion of the previous parts.

Thus, the latency of the pipeline for the forward- and backward-propagation can be expressed as:

$$L_{fwd} = max_{p=2}^{P-1}(procT_p^{fwd}(n_p)) \tag{1}$$

$$L_{back} = max_{p=2}^{P-1}(procT_p^{back}(n_p)) \tag{2}$$

where $procT_p^{fwd}(n_p)$ and $procT_p^{back}(n_p)$ are the processing times for the `fwd()` and `back()` steps for a single batch on the compute node responsible for M_p. Then, the average batch processing time when the *pipeline is full* (every node has at least one `fwd()` and one `back()` task), is

$$procT_{batch}^{pipefull} = L_{fwd} + L_{back} \tag{3}$$

We assume that nodes perform computing and communication tasks concurrently, which is reasonable for modern platforms.

Cold start & first batch. When a new training (global) epoch starts, the pipeline is initially empty. Therefore, the time for the very first batch to get processed by the pipeline, and for the pipeline to get filled with `fwd()` and `back()` tasks, is

$$procT_{batch}^{pipeempty} = \sum_{p=2}^{P-1} procT_p^{fwd}(n_p) + \sum_{p=2}^{P-1} procT_p^{back}(n_p)$$

This is the result of assuming that the compute nodes are idle when processing the `fwd()` and `back()` tasks for the first batch. Although such an assumption is true for the `fwd()` tasks, this is not fully valid for the `back()` tasks; when these arrive, the pipeline is filled with the `fwd()` tasks. Yet, for simplicity, this contention is ignored, which is acceptable since the empty phase is a minimal period of the training process. Hence, the equation can be considered an optimistic lower bound. Likewise, there is an additional delay before the first batch starts being processed by the pipeline, which is the time needed by the first data owner to perform the local `fwd()` task for M_1 and send the activations to the compute node n_2 responsible for M_2. This delay is estimated as

$$startT_{batch}^{first} = \frac{1}{K}\left(\sum_{k=1}^{K} procT_1^{fwd}(d_k) + commT^{fwd}(d_k, n_2)\right)$$

where $commT^{fwd}(d_k, n_2) = \frac{data_{1,2}^{fwd}}{bw_{d_k,n_2}}$ is communication delay for the transfer of the activations and $data_{1,2}^{fwd}$ is the respective amount of data that needs to be transferred between d_k and n_2. The reason for using the average over all data owners is that we do not know beforehand which one will send the first batch to the pipeline.

End of epoch. There is a similar delay for the last batch of the epoch after this has been processed by the pipeline, to transfer the gradients from n_2 to the last data owner and to perform the local `back()` for M_1, which is not overlapped by other processing tasks. This delay is equal to

$$endT_{batch}^{last} = \frac{1}{K}\left(\sum_{k=1}^{K} commT^{back}(n_2, d_k) + procT_1^{back}(d_k)\right)$$

Total delay for all batches. Like in FL, and also in SFL, all K data owners train the model by feeding each batch r consecutive iterations (each such iteration corresponds to a so-called *local epoch*). The total time needed for this is

$$\begin{aligned} T_{batch}^{all,multi-hop} &= startT_{batch}^{first} + procT_{batch}^{pipeempty} \\ &+ (r \cdot B \cdot K - 1) \cdot procT_{batch}^{pipefull} + endT_{batch}^{last} \end{aligned} \tag{4}$$

One-hop delay. The eq. (4) is used for multi-hop SFL, but can also be used for the single-hop SFL. Firstly, the core delay, i.e., $procT_{batch}^{pipefull}$, becomes, $procT_2^{fwd}(n_2) + procT_2^{back}(n_2)$. Further, the delay for the "cold start" can be ignored. While the delays for the first and last batch remain the same. Hence the total delay for all batch updates in one epoch in single-hop SFL is

$$\begin{aligned} T_{batch}^{all,one-hop} &= startT_{batch}^{first} + r \cdot B \cdot K \cdot (procT_2^{fwd}(n_2) \\ &+ procT_2^{back}(n_2)) + endT_{batch}^{last} \end{aligned}$$

Aggregation and global epoch delay. The aggregation of the model parts corresponding to each data owner is performed *independently* for each of the intermediate parts $M_p, 2 \leq p \leq P-1$ by the compute node n_p responsible for that part. Notably, no communication is required for this between the data owners and/or the compute nodes. However, the aggregation for the first and last parts (hosted on the data owners), does require extra communication. We assume, similarly to `FedAvg`, that a designated node c_{aggr} is used exclusively for this purpose. When a data owner completes the last batch, it sends the model updates to c_{aggr}. Note that, for the large majority of the data owners, this communication takes place while the pipeline is processing other tasks. Therefore, this communication largely overlaps with the training phase and does not introduce any significant additional delay. Nevertheless, for c_{aggr} to start the actual aggregation, it needs to wait until it receives the model updates from the last data owner. Let $data_{M_1}^{aggr}$ and $data_{M_P}^{aggr}$ be the amount of data each data owner needs to exchange with c_{aggr} for the first and last part of the model, respectively. Also, let

$commT^{aggr}_{M_1,M_P}(d_k, c_{aggr}) = \frac{data^{aggr}_{M_1}+data^{aggr}_{M_P}}{bw_{c_{aggr},d_k}}$ be the delay for the data transfer between d_k and c_{aggr}. Then, the aggregation delay is

$$T_{aggr} = \frac{1}{K}\left(\sum_{k=1}^{K}(commT^{aggr}_{M_1,M_P}(d_k, c_{aggr}))\right) + procT^{aggr}(M_1, M_P) + \sum_{k=1}^{K} commT^{aggr}_{M_1,M_P}(c_{aggr}, d_k) \quad (5)$$

The first term captures the delay for transmitting the model updates from the last data owner to c_{aggr}. Since this could be any of the data owners, the delay is estimated using the average communication delay overall data owners. The second term is the time needed by c_{aggr} to aggregate all M_1 and M_P updates and produce the respective global updated parts. Finally, the third term is the delay in the transmission of the updated model parts from c_{aggr} back to the data owners. Based on all the above, the total delay for the completion of a global epoch, including aggregation, is equal to

$$T_{tot} = T^{all}_{batch} + T_{aggr} \quad (6)$$

4 Optimization problem

The mathematical model we have presented not only can be used as a training delay estimation tool but it can also be extended into an optimization problem that solves the joint problem of cut layer decision and model part assignment. In detail, given the sets of $\mathcal{K}$ data owners and $\mathcal{N}$ compute nodes, the split points and assignment of the intermediate model parts can be optimized by minimizing the latency of the pipeline, expressed in eq. (3). But, we allow data owners to determine the first and last cut layers, as this decision depends on users' preferences regarding resource allocation.

Let $\mathcal{S}, |\mathcal{S}| = S$ be the set of layers of the model M. Also, we define the decision variable $\mathbf{x} = (x_{ij} \in \{0, 1\}, (i \in \mathcal{N}, j \in \mathcal{S}^*))$, where $\mathcal{S}^* \subseteq \mathcal{S}, |\mathcal{S}^*| = S^*$ is the subset of the layers of the intermediate model parts; does not contain the layers of M_1 and M_P. If layer j is offloaded into compute node c_i, then $x_{ij} = 1$, otherwise $x_{ij} = 0$.[2]

Constraints. Each layer can only be offloaded into one compute node, and each compute node receives at least one layer,

$$\sum_{i=1}^{N} x_{ij} = 1, \ \forall j \in \mathcal{S}^* \text{ and } \sum_{j=1}^{S^*} x_{ij} \geq 1, \ \forall i \in \mathcal{N} \quad (7)$$

The compute nodes handle model parts of sequent layers,

$$\sum_{k=2}^{j} x_{ij}(x_{i,k-1} - 2x_{ij} - x_{ik}) \leq 0, \ \forall j \in \mathcal{S}^*, \forall i \in \mathcal{N} \quad (8)$$

Further recall that mem_{c_i} denotes the available memory of compute node c_i, and that the compute node stores a distinct copy of the model part for each data owner. Let μ_j represent the memory demand for layer j. Therefore,

$$K\sum_{j=1}^{S^*} x_{ij}\mu_j \leq mem_{c_i}, \ \forall i \in \mathcal{N} \quad (9)$$

[2]The decision variable has a double definition, it defines the "borders" of each model parts (i.e., the cut layers) and the assignment of them into the compute nodes.

ID	Platform	CPU	Memory	ResNet	VGG
d_1	RPi 4 B	Cortex-A72 (4 cores)	4GB	91.9(1.8)	71.9(1)
d_2	RPi 3 B+	Cortex-A5 (4 cores)	1GB	no memory	
VM	CentOS 7.9	8-core virtual CPU	16GB	2(0.18)	3.6(0.1)

Table 1: Testbed nodes and average batch update delay in seconds (standard deviation in brackets); batch size is 128.

Then, the Equations (1) and (2) can be updated accordingly,

$$L_{fwd} = max_{i=1}^{N}\{\sum_{j=1}^{S^*} x_{ij} procT^{fwd}_{j}(n_{c_i})\} \quad (10)$$

$$L_{back} = max_{i=1}^{N}\{\sum_{j=1}^{S^*} x_{ij} procT^{back}_{j}(n_{c_i})\} \quad (11)$$

Objective. The objective is to find the splitting points and model assignments, that minimize the pipeline delay. Thus, we construct the following optimization problem

$$\mathbb{P} \ : \ \underset{\mathbf{x}}{\text{minimize}} \quad procT^{pipefull}_{batch}$$
$$\text{s.t. } (3), (7)-(11), \text{ and } \mathbf{x} \in \{0,1\}^{N\times S^*}.$$

The empirical analysis in Sec. 5 shows that $\mathbb{P}$ is a lightweight problem that can be solved with any ILP solver with an overhead less than 1 sec when considering 5 hops.[3]

5 Experiments

In this section, we present the evaluation of the proposed mathematical model for estimating the training delay. As indicative ML models, we use ResNet-101 [4] and VGG-19 [11] trained with the CIFAR-10 [6] dataset. Each data owner gets 16 batches ($B = 16$) of 128 samples. In SFL and FL, smaller batch sizes and simpler NN models are typically used. Here, we aim to introduce a more ambitious system to create challenging scenarios for evaluating the proposed model. Also, there are $r = 2$ local epochs. Notably, the selection of B and r is arbitrary as they are hyper-parameters of the ML model, and do not affect the estimation model.

Testbed. A physical testbed was used to measure the performance of the model. As data owner nodes, we use two different Raspberry Pi devices, while for the compute nodes we use VMs running in a private cluster. The communication between data owners and compute nodes is via VPN over WiFi and the public Internet. Their characteristics and the average time for one batch update for the full model are given in Table 1. Note that d_2 cannot support on-device training due to memory limitations.[4] Note, the two data owner devices have notable different characteristics, i.e., d_1 is significantly faster than d_2 (as discussed in Fig. 4). This builds more generalized experiments, by simulating system heterogeneity.

Emulation of numerous data owners. Since we only have a few devices at our disposal, it is not possible to perform large-scale experiments. Instead, we emulate a large number of data owners using additional nodes in our cluster. To this end, we profile the

[3]We also have made available the corresponding source code: https://github.com/jtirana98/MultiHop-Federeated-Split-Learning

[4]This is not an issue for our analysis since we use SFL, which reduces the memory and compute demands at the data owners, by splitting the model.

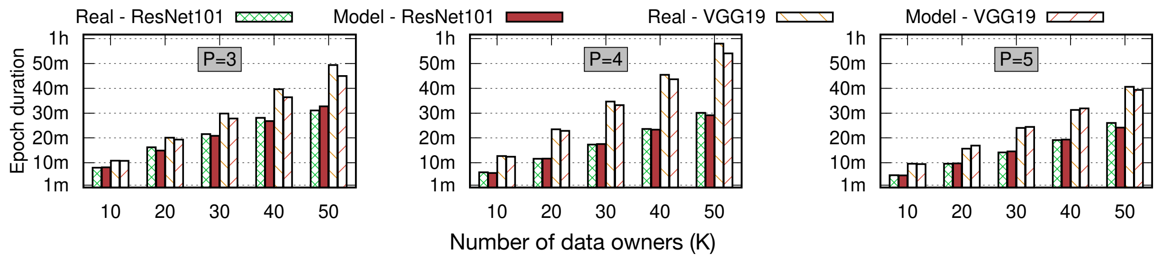


Figure 3: Average training performance of real experiments for d_1 vs the performance estimated by the model.

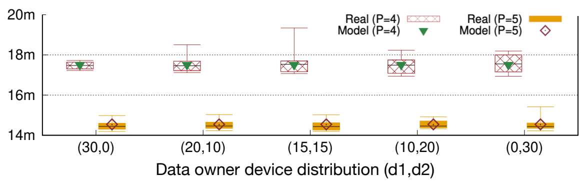


Figure 4: Model validation for 30 heterogeneous data owners devices. The distribution notation (q_1, q_2) means that there are q_1 devices of type d_1 and q_2 devices of type d_2.

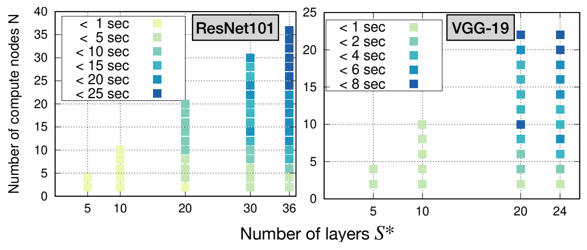


Figure 5: Computing time for solving $\mathbb{P}$ with Gurobi.

`fwd()` and `back()` tasks on d_1 and d_2 for ResNet-101 and VGG-19 and measure the throughput between d_1 and d_2 and the VMs. These measurements are then used to run multiple data owner processes on the VMs of the cluster. To mimic the behavior of the resource-constrained devices, processing and communication delay are artificially slowed down as needed.

Model validation. We measure the epoch time of the training using the testbed and compare it with the estimate obtained via our model. Fig. 3 shows the results with one ($P = 3$), two ($P = 4$) and three ($P = 5$) compute nodes, for up to 50 data owners of type d_1. As can be seen, the model is close to the real results, with an average absolute error of 3.14% over all experiments for ResNet-101 and 3.86% for VGG-19. Note, that when there are fewer splits and, as a result, the compute nodes are assigned larger model parts, the delay estimation of the model is smaller than the actual measurements. There is a similar deviation as the number of data owners increases. This is a side-effect of the memory pressure in the compute nodes (not captured by the model). *Still, the model is sufficiently accurate to serve as a tool for investigating a wide range of scenarios.*

Further, we run experiments with a heterogeneous distribution of data owners and compare the measured epoch delay with the estimates of our model. Fig. 4 shows the results of training ResNet-101 with two and three compute nodes for 30 data owners, as the portion of d_1 vs d_2 devices varies. A single batch of d_2 needs roughly 7.25 seconds to complete the `fwd()` and `backward()` tasks of the first and last model part, while d_1 needs about 3.15 seconds for this, thus is 2.3x faster. *We notice that the real delay is very close to the model estimates.* Note that, the median of the real epoch delay has a small fluctuation, up to 1.98 seconds for three compute nodes ($P = 5$) and 5 seconds for two ($P = 4$). This is a reasonable side effect of running experiments on a shared infrastructure like our cluster or in the cloud, where VMs can occasionally experience a slowdown in their execution due to multi-tenancy. However, this sets an even more realistic set-up as such effects are expected in edge-cloud settings.

After having validated the accuracy of the mathematical model we proceed with the evaluation of the joint problem of split point selection and model part assignment described in Sec. 4.

Overhead for splitting decision. We utilize Gurobi a well-known ILP solver, to solve problem $\mathbb{P}$. In Fig. 5 we conduct a sensitivity analysis of the problem's size, which is proportional to the number of possible splits S^* and the number of compute nodes N. Fig. 5 shows the computing time of the solver as the values of S^* and N change. We measure the computing time using a compatible MacOS laptop with Apple M1 chip and 16 GB of memory. We notice that the computing time is more sensitive to the number of compute nodes, which can be seen in the left plot (using ResNet-101) in which S^* has the larger values, and the computing time gets greater as N ascends. Similarly, this effect is seen in the VGG-19 model (right plot). Nevertheless, the optimizing cost remains negligible, considering that this is a one-time overhead since $\mathbb{P}$ and can be solved during the offline period before the actual training starts or when transiting between epochs. In summary, for a typical number of hops (up to 5) the additional overhead is less than 1 sec.

The importance of $\mathbb{P}$. Fig. 6 compares the epoch's delay when using the proposed splitting-allocation mechanism vs. a self-designed benchmark approach.[5] The results in Fig. 6 show that when using the optimizer the improvements per epoch can be up to 8.5% for the ResNet-101 and up to 19% for the VGG-19 (which is a more challenging model to manage manually [14]).

[5]We divide the total computing time of the intermediate part by the number of compute nodes. Then, we manually assign to the compute nodes subsequent layers that sum up to a latency close to the computed one (i.e., trying not to exceed it). This is not always feasible, as the computing demands for each layer can be arbitrary [5].

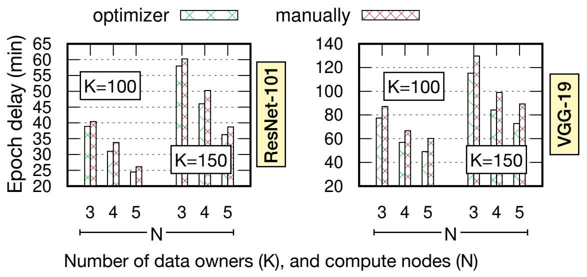


Figure 6: Epoch duration when split points are optimized using $\mathbb{P}$ or manually selected, for different K, and N.

6 Related Work

The proposed mathematical model contributes to the SFL research in two different manners. First, it provides a new and novel tool that can be used for estimating the training time of multi-hop SFL with PP. The benefits of PP are (i) accelerated training and (ii) support for larger models. The use of multi-hop SFL and PP is starting to evolve gradually. The first such approach is [13], while a more recent one is RingSFL [10]. Also, in [2] the nodes create clusters to help each other, by offloading model parts to each node. However, each cluster is constructed to help one single data owner, which does not fully exploit PP, similarly case in [7]. Further, [19], considers PP with one hop. While, FedSL [1] supports multi-hop for recurrent NNs. Even though multi-hop SFL and PP are starting to be widely used, an estimation tool has not yet been established. For example, the model in RingSFL is strictly for finding the inference length. Moreover, in [3] a set of estimation models is proposed but none of them considers PP in SFL, and a lot of over-simplifications are made, e.g., the computing and communication delay are not expressed as factors of the resources and per-layer cost, like in our model. Thus, we believe that our mathematical model emerges right on time when the exploration of multi-hop SFL is growing. To this end, we would like to highlight that the scope of the proposed work is to provide a new estimation tool for multi-hop SFL and not to propose a new SFL design. Hence, other examples of multi-hop SFL e.g. [10, 13] are orthogonal to the contributions of our work.

Further, the proposed work provides a simple optimization mechanism. In SL, one of the most crucial decisions is identifying the cut layers since it directly affects the training delay. In existing research work, the most commonplace considered system is the one with multiple data owners and one compute node (i.e., one-hop) [9, 18]. Typically, in these works an optimization problem is built according to several parameters (e.g., energy consumption, delay, etc.). Alternatively, [17] uses Reinforcement Learning to find the best split, but it is not suitable for large-scale cases. Only CoopFl [16] considers the case of multiple compute nodes but only in the horizontally scaled SFL context (not in the multi-hop configuration). To this end, to the best of our knowledge, we have proposed the first generalized optimization model for SFL (one- or multi-hop).

7 Conclusions

In this work, we designed a general mathematical model for estimating the expected performance of single- or multi-hop SFL with an error smaller than 3.86%. This model can estimate the expected training time and serve as a tool for tuning key system parameters (e.g., number of hops, batch size, etc.). Moreover, the proposed tool can be utilized by the research community as an evaluation tool to help them conduct assessments of their frameworks, at low cost. Additionally, we have formulated a lightweight optimization problem for the cut-layer decision in multi-hop SFL. To the best of our knowledge, this is the first work to propose such a general model for SFL, accommodating all possible SFL splitting configurations.

References

[1] Ali Abedi and Shehroz S Khan. 2024. FedSL: Federated split learning on distributed sequential data in recurrent neural networks. *Multimedia Tools and Applications* 83, 10 (2024), 28891–28911.

[2] Zhipeng Cheng, Xiaoyu Xia, Minghui Liwang, Xuwei Fan, Yanglong Sun, Xianbin Wang, and Lianfen Huang. 2023. CHEESE: distributed clustering-based hybrid federated Split learning over edge networks. *IEEE Trans. on Parallel and Distributed Systems* (2023).

[3] Flavio Esposito, Maria A Zuluaga, et al. 2022. DTS: A simulator to estimate the training time of distributed deep neural networks. In *2022 30th International Symposium on MASCOTS*. IEEE, 17–24.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proc. of the IEEE conference on computer vision and pattern recognition*. 770–778.

[5] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.

[6] Alex Krizhevsky, Geoffrey Hinton, et al. 2009. Learning multiple layers of features from tiny images. In *Toronto, ON, Canada*.

[7] Zheng Lin, Wei Wei, Zhe Chen, Chan-Tong Lam, Xianhao Chen, Yue Gao, and Jun Luo. 2024. Hierarchical split federated learning: Convergence analysis and system optimization. *arXiv preprint arXiv:2412.07197* (2024).

[8] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *AISTATS*. PMLR, 1273–1282.

[9] Eric Samikwa, Antonio Di Maio, and Torsten Braun. 2022. ARES: Adaptive Resource-Aware Split Learning for Internet of Things. *Computer Networks* 218 (2022), 109380.

[10] Jinglong Shen, Nan Cheng, Xiucheng Wang, Feng Lyu, Wenchao Xu, Zhi Liu, Khalid Aldubaikhy, and Xuemin Shen. 2023. Ringsfl: An adaptive split federated learning towards taming client heterogeneity. *IEEE Transactions on Mobile Computing* 23, 5 (2023), 5462–5478.

[11] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[12] Chandra Thapa, Pathum Chamikara Mahawaga Arachchige, Seyit Camtepe, and Lichao Sun. 2022. Splitfed: When federated learning meets split learning. In *Proc. of the AAAI Conference on Artificial Intelligence*, Vol. 36. 8485–8493.

[13] Joana Tirana, Christodoulos Pappas, Dimitris Chatzopoulos, Spyros Lalis, and Manolis Vavalis. 2022. The role of compute nodes in privacy-aware decentralized AI. In *Proc. of EMDL*. 19–24.

[14] Joana Tirana, Dimitra Tsigkari, George Iosifidis, and Dimitris Chatzopoulos. 2025. Minimization of the Training Makespan in Hybrid Federated Split Learning. *IEEE Transactions on Mobile Computing* (2025).

[15] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2018. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564* (2018).

[16] Zhiyuan Wang, Hongli Xu, Yang Xu, Zhida Jiang, and Jianchun Liu. 2023. CoopFL: Accelerating federated learning with DNN partitioning and offloading in heterogeneous edge computing. *Computer Networks* 220 (2023), 109490.

[17] Di Wu, Rehmat Ullah, Paul Harvey, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. 2022. Fedadapt: Adaptive offloading for iot devices in federated learning. *IEEE Internet of Things Journal* (2022).

[18] Wen Wu, Mushu Li, Kaige Qu, Conghao Zhou, Xuemin Shen, Weihua Zhuang, Xu Li, and Weisen Shi. 2023. Split learning over wireless networks: Parallel design and resource management. *IEEE Journal on Selected Areas in Communications* 41, 4 (2023), 1051–1066.

[19] Zihan Zhang, Philip Rodgers, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. 2024. PiPar: Pipeline parallelism for collaborative machine learning. *J. Parallel and Distrib. Comput.* 193 (2024), 104947.