CycleSL: Server-Client Cyclical Update Driven Scalable Split Learning

Anonymous authors
Paper under double-blind review

Abstract

Over the past few years, split learning has developed by leaps and bounds due to the shift towards distributed computing and the demand for data privacy. To increase the scalability of sequential split learning, scalable variants such as parallel split learning and split federated learning have been proposed, which often entail huge computation and memory consumption on the server side, limiting thus their scalability. Moreover, former aggregation-based methods generally converge with inferior rate and quality due to factors such as client drift and lag, whilst existing aggregation-free methods cannot really benefit from parallelism. In this paper, we present a novel aggregation-free split learning paradigm termed ***CycleSL***, which can be integrated into existing algorithms to boost model performance while imposing less resource consumption. Inspired by alternating coordinate descent, CycleSL models the training task on the server side as a standalone higher-level machine learning task and updates the server and client in cyclical turns through the reuse of smashed data. Benefiting from feature resampling and alternating gradient steps, CycleSL has great potential to advance model performance and robustness. We integrate CycleSL into previous algorithms and benchmark them on four publicly available datasets with non-iid data distribution and partial client attendance. Our results show that CycleSL can notably improve model performance and convergence.

1 Introduction

As a distributed collaborative machine learning paradigm, split learning (SL, Gupta & Raskar (2018); Vepakomma et al. (2018)), which is also called split neural network (SplitNN), has recently gained strong momentum given the rapid development of distributed computing resources and the ever-growing demand for data privacy. In this paper we limit our discussion to horizontal SL, which means the samples of clients share the same feature space but are different in the sample space. Compared to centralized learning Soykan et al. (2022), where both data storage and model training occur in a centralized manner, in SL, data is distributed across a set of clients, and the training load is amortized between server and clients. A similar concept is federated learning (FL, McMahan et al. (2017)), where each client holds a complete model copy and trains its local model using its individual data while a server periodically aggregates client models. SL is different from FL, as in SL, clients only train their models up to a cut layer and send extracted features, which are also called smashed data, to a server, while the server completes the rest of the training without requiring raw data and sends gradients back to clients for their local update. Through such a procedure, the training load can be shared among participating entities without burdening one side too much, and primary data privacy can be guaranteed, given that no data or model sharing is needed. Moreover, since SL shifts a proportion of computation load to from clients to server side, it can be more practical in collaborative computing in comparison to FL.

The canonical SL happens sequentially, meaning the server only pairs with one client each time. To be the next one being serviced without a cold start, a client needs to retrieve the latest trained client part model from the last coupled client, either through the server or a trusted third party or client-to-client peer sharing. That said, the model trained in sequential SL is not essentially different from a model trained in the centralized manner, except all hidden features in a client-to-server mini-batch come from the same data

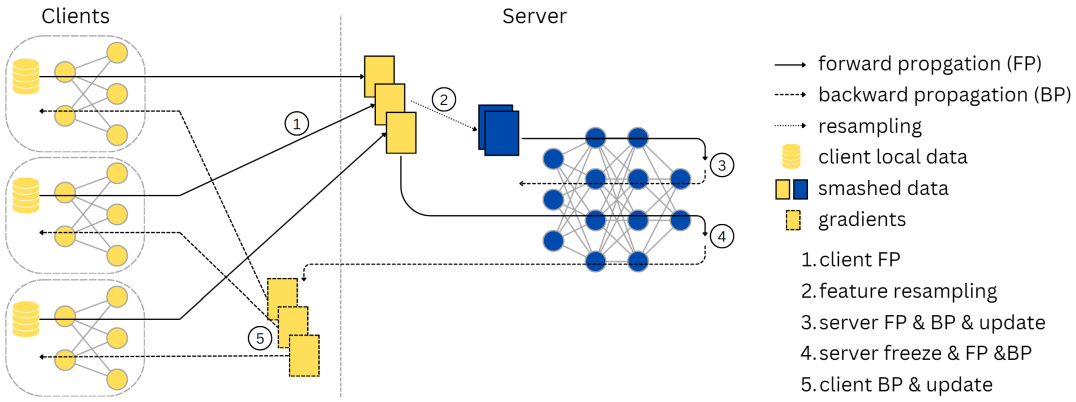


Figure 1: The CycleSL pipeline. After collecting smashed data from clients, CycleSL first forms a global feature dataset on the server side, which is regarded as input samples for a higher-level training task. Then CycleSL resamples features from the dataset to train the server model. Only after the server model is updated, the original feature batches are reused to compute gradients using the latest server model. Lastly, the gradients are sent back to clients for their local update.

holder. This way, sequential SL is on par with centralized learning regarding model performance but suffers from high latency due to poor scalability. To increase the scalability of sequential SL, many parallelized variants of SL, which are often realized in combination with FL, have been introduced, such as parallel split learning (PSL, Jeon & Kim (2020); Joshi et al. (2021); Lin et al. (2024b)), split federated learning (SFL or SplitFed, Thapa et al. (2022); Lin et al. (2024a)), federated split learning (FSL, Turina et al. (2021); Zhang et al. (2023)), and other forms of combinations Han et al. (2021); Pal et al. (2021); Abedi & Khan (2023); Wu et al. (2023). These methods generally approach scalability by duplicating the server part model or even the server itself, which could strongly stress the server. Further, due to the integration of FL, they often unavoidably suffer from FL problems such as client drift Karimireddy et al. (2020); Charles & Konečnỳ (2021); Wang et al. (2021). Consequently, they may yield worse convergence quality and rate than sequential SL, which can be costly to compensate under distributed learning setup, especially in cross-device scenario Kairouz et al. (2021).

In this paper, we propose a novel aggregation-free SL paradigm coined ***CycleSL***, which can be integrated into other scalable SL algorithms to improve model performance and convergence rate while imposing even less resource burden on the server side, especially compared to aggregation-based methods. Firstly, inspired by the model-as-sample strategy Wang et al. (2024), CycleSL models the training task on the server side as a standalone higher-level machine learning task with the smashed data from clients as input samples, and resamples mini-batches from the collected smashed data to counteract data heterogeneity. Then, CycleSL adapts the inspiration from alternating (block) coordinate descent Luenberger et al. (1984) and performs one step further on the server side, namely updates the server part and client part models in cyclical turns, rather than in the common end-to-end manner Glasmachers (2017) in the same gradient backward flow. A core advantage of such server-client alternating update is that it could bring in the benefits of coordinate descent solutions and hence further counteract client drift. The main contributions of this work are four-fold:

- We present a perspective to model the server side training in split learning as a higher-level machine learning task and regard the hidden representations from clients as input samples in a feature-as-sample manner.
- We propose server-client cyclical update in split learning, which follows the alternating update strategy of coordinate descent and can thus benefit from it. In this regard, we give a heuristic toy example for the potential implicit regularization effect of our method.
- We introduce CycleSL, an aggregation-free split learning paradigm that realizes the aforementioned two ideas and can be combined with other scalable split learning algorithms. We integrate CycleSL

into three recent methods, including PSL, SFL, and SGLR, and suggest CyclePSL, CycleSFL, and CycleSGLR accordingly.

- We benchmark the aforementioned methods on four publicly available datasets including FEMNIST, CelebA, Shakespeare, and CIFAR-100, with non-iid data distribution across clients and a 5% client attendance rate. Our results show that the cycle-version methods remarkably outperform their originals on test performance.

To our best knowledge, we are the first to suggest the server side training as a standalone task and to harness coordinate descent for server-client alternating update in SL. Our work addresses a new research gap in this regard. We also publish our code for reproducibility[1]. By following our instructions, our results can be replicated up to numerical error.

2 Related work

2.1 Scalable split learning

To improve the scalability of vanilla sequential SL, model parallelism has been introduced into SL, which allows multiple clients to collaborate with the server simultaneously. The scalability is often realized through model duplicates on the server side, with one server model clone pairing with one client during training, whose parameters or gradients are later aggregated with the help of FL algorithms such as FedAvg McMahan et al. (2017) or FedOpt Reddi et al. (2020). For instance, PSL Jeon & Kim (2020); Joshi et al. (2021) enables multi-client connection by replicating the server part model and periodically averaging these model copies. SFL Thapa et al. (2022), especially its first variant, SFLV1, does exactly the same on the server side while also engaging a trusted third party (not necessarily the server itself) to aggregate the client part models. In contrast, FSL Turina et al. (2021) engages multiple servers rather than multiple models on one server, making the server-client connection more flexible. CPSL Wu et al. (2023) clusters clients into groups to reduce the latency caused by sequential processing. Unlike the methods above, which generally work for most feed-forward models, FedSL Abedi & Khan (2023) is specialized for recurrent neural networks.

A direct (negative) impact of parallelism based on model duplication is the extra computation and memory consumption for server, particularly when the cut layer is shallow and most of the model parameters are kept on the server side. As a consequence, the scalability can be strongly limited. Moreover, as most scalable SL methods rely on federated aggregation, they often unavoidably suffer from FL problems. One of the most challenging aspects is client drift Karimireddy et al. (2020); Charles & Konečnỳ (2021); Wang et al. (2021), which describes the phenomena that local models converge towards local optima instead of global optimum due to factors like data heterogeneity and data imbalance. This problem is especially challenging in cross-device scenarios Kairouz et al. (2021), where a single user with a personal device like smartphone or tablet and his personal data forms a client. Since each individual can have a unique data distribution, their local optima could diverge to a large extent. The case could be even worsened for some SL algorithms like PSL, where only the server model duplicates are aggregated while the client part models remain unsynchronized. As a result, aggregation-based SL methods often converge with lower rate and worse quality than sequential SL. The aggregation-free variants like SFLV2 (the second version of SFL Thapa et al. (2022)) address this challenge by engaging only one server model and processing all clients one by one, which is in essence not different from sequential SL on the server side and thus does not take advantage of parallelism. Client drift can be further complicated by partial participation McMahan et al. (2017); Kairouz et al. (2021), which means only a subset of clients can connect to the server in a round. This is especially an issue for methods without model orchestration on the client side, like PSL, since clients who lag behind due to absence in the last rounds may have a cold start with a less or even untrained local model.

More recently, a growing body of research aimed to address the aforementioned issues. For instance, EPSL Lin et al. (2024b) aggregates the last layer gradients before propagation to reduce the computation burden on the server side. SGLR Pal et al. (2021) averages gradients before sending them back to clients to mitigate client divergence without client model aggregation. It further sets different learning rates for each end to better

[1] `https://github.com/AnonymWriter/CycleSL`. This repository is anonymized for review.

adapt them. In Han et al. (2021), the authors reduced communication by utilizing localized loss functions on each end. A common limitation of these methods lies in that they often still rely on model duplication and aggregation, and thus their improvement can be strongly restricted in practice.

2.2 Coordinate descent

Coordinate descent is a classical optimization technique for multivariate functions. Given a multivariate function, coordinate descent approaches its minimum by optimizing along each dimension at one time alternatively rather than along all directions jointly. This procedure is repeated iteratively, (randomly) cycling through all the variables. According to Wright (2015), coordinate descent has multiple advantages, such as simplicity, scalability, and efficiency for high dimensions and sparsity. Though less studied, the alternating optimization mechanism of coordinate descent can result in a biased solution that resembles the effects of implicit regularization under certain conditions Nakamura et al. (2021); Zhao et al. (2022). The idea of coordinate descent and its application in machine learning can be traced back to the early days, covering various domains such as linear sparse problems Daubechies et al. (2004), Lasso and Ridge regularization Fu (1998), support vector machines Platt (1998), and matrix factorization Zhou et al. (2008). From the perspective of deep learning, techniques like layer-wise training Bengio et al. (2006); Hinton et al. (2006); Palagi & Seccia (2020) can be regarded as its applications. Though serving different purposes, regularization methods like dropout Hinton et al. (2012) also share some analogies with coordinate descent. More recently, (block) coordinate descent has been studied as gradient-free optimizer Carreira-Perpinan & Wang (2014); Zhang & Brand (2017); Zeng et al. (2019) as an alternative to gradient-based and second-order methods such as stochastic gradient descent (SGD) and Newton's method.

3 Methodology

Given the problems of former aggregation-based and -free scalable SL algorithms and the benefits of coordinate descent, we introduce ***CycleSL***, a novel aggregation-free SL mechanism that models the training of the server part model as a standalone higher-level task and applies feature resampling to counteract client drift, and updates the server and client part models in cyclical turns, following the alternating update strategy of (block) coordinate descent instead of in the common end-to-end pattern. In this section, we stick to the following notations. In a SL scenario with $N \in \mathbb{N}^+$ clients and $T \in \mathbb{N}^+$ epochs, let θ_C be the client model and θ_S be the server model, respectively. We use superscript $t \in [T]$ to denote epoch and subscript $i \in [N]$ to index client. We mark mini-batches using $\mathcal{B}$, with superscripts x for samples, f for feature, and g for gradients. Further we represent the objective function as $\mathcal{L}$ and the learning rate as η. The pipeline of CycleSL is depicted in Figure 1, and its pseudo-code is proviced in Algorithm 1. As CycleSL is aggregation-free, it requires significantly less computation and memory resources than other aggregation-based methods. An overarching comparison between CycleSL and other SL paradigms regarding their mechanisms and costs is given in Table 1.

3.1 Higher-level task with feature resampling

In the following, we illustrate our algorithm in detail, starting with the idea of standalone server-level task. We observe that it is, in principle, hard to refrain aggregation-based scalable SL methods from the high resource occupation due to their nature. One exception in this regard is SFLV2 (the second version of SFL), where there is only one server with one model servicing all participating clients one by one, and it is, hence, aggregation-free. However, in substance SFLV2 conducts sequential SL on the server side and does not benefit much from client parallelization. Moreover, both existing aggregation-based and -free methods are often inevitably influenced by client drift, as all smashed data in a client-to-server batch is extracted with the same client model from samples of the same data holder. For this reason, the server model can be biased towards a client's local optima in each batch step. To alleviate this issue, we regard the smashed data sent by clients to server as input samples for a standalone machine learning task at a higher level. A similar idea was proposed in TurboSVM-FL Wang et al. (2024), where class embeddings from client models are taken as samples in order to fit a secondary-level support vector classifier. To prevent the client-binding batch problem, CycleSL resamples the collected smashed data into random batches that are not no longer bound

Algorithm 1 The CycleSL framework.

Input: clients $i \in [N]$, client local datasets D_i, client part models θ_i, server part model θ_S, objective function $\mathcal{L}$, learning rate η, SL rounds T.

ClientFP (client i): # *parallelizable*
client i samples one mini-batch samples $\mathcal{B}_i^x \subseteq D_i$
client i computes features $\mathcal{B}_i^f \leftarrow \theta_i^t(\mathcal{B}_i^x)$
return $\mathcal{B}_i^f$

ClientBP (client i, gradients $\mathcal{B}_i^g$): # *parallelizable*
client i computes gradients regarding θ_i^t: $\nabla_{\theta_i^t}\mathcal{B}_i^g$
client i locally updates $\theta_i^{t+1} \leftarrow \theta_i^t - \eta\nabla_{\theta_i^t}\mathcal{B}_i^g$

ServerGrad (features $\mathcal{B}^f$, freeze):
if freeze is true **then**
  server disables gradient tracking for θ_S^{t+1}
  server computes loss $\ell \leftarrow \mathcal{L}(\mathcal{B}^f, \theta_S^{t+1})$
  server computes gradients regarding $\mathcal{B}^f$: $\nabla_{\mathcal{B}^f}\ell$
  return $\nabla_{\mathcal{B}^f}\ell$
else
  server enables gradient tracking for θ_S^t
  server computes loss $\ell \leftarrow \mathcal{L}(\mathcal{B}^f, \theta_S^t)$
  server computes gradients regarding θ_S^t: $\nabla_{\theta_S^t}\ell$
  server updates $\theta_S^{t+1} \leftarrow \theta_S^t - \eta\nabla_{\theta_S^t}\ell$
end if

Main:
initialize $\theta_1^0, \theta_2^0, ..., \theta_N^0$ and θ_S^0
for $t = 0, 1, ..., T-1$ **do**
  sample a subset of clients $S_t \subseteq [N]$
  for client $i \in S_t$ **do**
    $\mathcal{B}_i^f \leftarrow$ **ClientFP** (i)
  end for
  server forms feature-as-sample dataset $D_S^f \leftarrow \biguplus_{i \in S_t} \mathcal{B}_i^f$
  for server randomly resamples mini-batch $\mathcal{B}_S^f \subseteq D_S^f$ **do**
    ServerGrad ($\mathcal{B}_S^f$, false)
  end for
  for client $i \in S_t$ **do**
    $\mathcal{B}_i^g \leftarrow$ **ServerGrad** ($\mathcal{B}_i^f$, true) # *parallelizable*
    ClientBP (i, $\mathcal{B}_i^g$)
  end for
end for

to clients. Mathematically, this procedure can be expressed as follows. After receiving feature batches $\mathcal{B}_i^f$ from clients, previous SL paradigms generally updates server model using the batches directly:

$$\theta_S^{t+1} \leftarrow \theta_S^t - \eta\nabla_{\theta_S^t}\mathcal{L}_{i\sim[N]}(\mathcal{B}_i^f, \theta_S^t) \tag{1}$$

In contrast, CycleSL first combines these feature batches and forms a global feature dataset on the server side, then randomly resamples mini-batches B_S^f from the feature dataset and feeds them into the server model for training:

$$D_S^f \leftarrow \biguplus_{i\in[N]} \mathcal{B}_i^f, \theta_S^{t+1} \leftarrow \theta_S^t - \eta\nabla_{\theta_S^t}\mathcal{L}_{B_S^f\sim\mathcal{D}_S^f}(B_S^f, \theta_S^t) \tag{2}$$

Table 1: A comparison of SL methods regarding mechanisms and costs on the server side, given number of participating clients $N \in \mathbb{N}^+$. k is a factor dependent on CycleSL setup and generally $1 \leq k << N$.

	Sequential SL	Agg-based scal. SL	Agg-free scal. SL	CycleSL
Sequential pairing	yes	no	yes	no
Model aggregation	no	yes	no	no
Benefit from scalability	no	yes	no	yes
Computation intensity	$O(1)$	$O(N)$	$O(1)$	$O(1)$
Memory consumption	$O(1)$	$O(N)$	$O(1)$	$O(1)$
Latency	$O(N)$	$O(1)$	$O(N)$	$O(k)$

A further benefit of this procedure lies in the reduction of computation and memory cost, since the server part training is not aggregation-based. In addition, as the server part training is a complete standalone task by itself, more flexible configurations of hyperparameters, such as learning rate decay and normalization strategies, can be independently determined to improve model performance.

3.2 Server-client cyclical update

In the conventional SL pipeline, though the forward flow of hidden representations and the backward flow of gradients are cut in the middle and transmitted between entities, the client part and server part models are still updated with gradients computed from the same backward propagation, which follows an end-to-end Glasmachers (2017) training paradigm in general. Since the server side training is isolated from the client part as an independent task in CycleSL, we bring in the alternating update strategy of (block) coordinate descent and suggest the cyclical training of server and client models. More precisely, after the server model is optimized, CycleSL freezes the server part model so that no gradients will be computed for its parameters. Then, CycleSL reuses the smashed data from clients and feeds them into the already updated server model to calculate the gradients that will be sent back to clients. The server model shall be unfrozen afterwards. In the next step, clients update their local models with the received gradients, and the next round of SL will start. That said, the server and client part models can be regarded as two blocks in block coordinate descent, which are optimized independently and alternatively in rounds. The server-client cyclical update can also be interpreted as coordinate descent on composition function, where the complete network can be then considered as a function composition $\theta_{CS} = \theta_S \circ \theta_C$ with $\theta_{CS}(x) = \theta_S(\theta_C(x))$. Mathematically, whilst traditional SL algorithms update the client and server models "simultaneously", as they use the models at time step t to compute gradients for both the client and server models:

$$\theta_C^{t+1} \leftarrow \theta_C^t - \eta \nabla_{\theta_C^t} \mathcal{L}(\theta_C^t, \theta_S^t) \tag{3}$$

CycleSL updates the client model after the server model while respecting the latest update of the server model:

$$\theta_C^{t+1} \leftarrow \theta_C^t - \eta \nabla_{\theta_C^t} \mathcal{L}(\theta_C^t, \boldsymbol{\theta_S^{t+1}}) \tag{4}$$

There are multiple benefits of this update paradigm. First, since the gradients are only computed with respect to one set of parameters each time (either the server part model or the smashed data), while the other set does not trace gradient flow, the memory usage is not burdening. A further benefit lies in that the gradient computation regarding smashed data can be easily parallelized, e.g., with one frozen clone of the server model coupling with one client to accelerate the process. Thirdly, different from SFLV2, where each client is actually paired with a server in a different status due to sequential coupling, in CycleSL, all participating clients in one round couple with the identical server, thus alleviating the impact of client drift. Furthermore, the block-wise alternating freeze-and-train training style can potentially benefit from the merits of coordinate descent such as implicit regularization, as the optimized server model generally tends to yield lower loss value compared to the unoptimized one, which could in turn lead to smaller gradient steps on the client side. In this regard, we give a toy example of implicit regularization effect of CycleSL in the following. The purpose of this example is not to serve as strict proof but as a heuristic.

3.2.1 Toy example

Consider a simplified regression task in SL where both the client and server part models are a single linear layer with one neuron. We further ignore activation functions and bias terms. Then, for a sample point (x, y), its predicted value can be given as $\hat{y} = w_s w_c x$, where w_c and w_s are the parameters of client and server part models respectively. Consider mean squared error as a loss function, i.e., $\ell = (y - \hat{y})^2$. Traditionally, SL follows an end-to-end Glasmachers (2017) pattern, which means both w_c and w_s are updated in the same gradient backward flow: $w_s' \leftarrow w_s - 2\eta w_c x(w_s w_c x - y), w_c' \leftarrow w_c - 2\eta w_s x(w_s w_c x - y)$ where w_c' and w_s' are the updated parameters and $\eta > 0$ is learning rate. In contrast, CycleSL updates w_s and w_c one after another: $w_s' \leftarrow w_s - 2\eta w_c x(w_s w_c x - y), w_c' \leftarrow w_c - 2\eta w_s' x(w_s' w_c x - y)$.

Comparing the two update strategies, it is clear that both are identical with respect to server side training, while they respectively use the old and the new server models to update the client part model. We now compare the two gradient steps for the client model, namely $2\eta w_s x(w_s w_c x - y)$ and $2\eta w_s' x(w_s' w_c x - y)$, when approaching convergence. For simplicity, we limit our discussion to the case where all $w_c, w_s, x, y > 0$ and $w_s w_c x > y$. All other cases can be analyzed similarly. Since $w_s w_c x > y$, we expect that with a proper choice of η, w_s' is reduced for a decently small step during server training, i.e., $\frac{y}{w_c x} < w_s' < w_s$ such that $w_s' w_c x$ shrinks towards y. Observe the function $f(w_s) = w_s x(w_s w_c x - y)$ and its derivative $f'(w_s) = 2w_s w_c x^2 - xy = x(w_s w_c x - y + w_s w_c x)$. When approaching convergence, i.e. $w_s w_c x - y \to 0^+$, we have $f'(w_s) > 0$, which means $f(w_s)$ is increasing in its neighborhood. Thus for $\frac{y}{w_c x} < w_s' < w_s$ the following applies: $f(w_s') < f(w_s) \Leftrightarrow 2\eta w_s' x(w_s' w_c x - y) < 2\eta w_s x(w_s w_c x - y)$. That said, the alternating update strategy of CycleSL could result in a smaller gradient step on the client side compared to the conventional end-to-end paradigm when approaching convergence, which can serve as an implicit regularizer.

4 Experiments

We integrated CycleSL into three recent scalable SL algorithms, including PSL Jeon & Kim (2020); Joshi et al. (2021), SFL Thapa et al. (2022), and SGLR Pal et al. (2021), and introduced CyclePSL, CycleSFL, and CycleSGLR accordingly. Particularly, CyclePSL is, in essence, identical as described in the Algorithm 1. Counting in the two versions of SFL (SFLV1 and SFLV2) and FedAvg McMahan et al. (2017), We benchmarked in total eight algorithms on four publicly available datasets. Our experiments were conducted on a cluster with AMD EPYC 7763 64-Core and NVIDIA A100 80GB PCIe $\times$ 4. To ensure the robustness and reliability of results, we replicated the experiments over 5 random seeds from $\{0, 1, 2, 3, 4\}$ and reported mean $\pm$ std for each metric. Our code is published [1] to promote reproducibility.

4.1 Datasets and tasks

We utilized four datasets from LEAF Caldas et al. (2018) and FL-bench Tan & Wang (2023), both of which are standardized benchmark frameworks for FL. We chose these platforms because SL is closely related to FL, and the platforms guarantee a great level of reproducibility by providing baseline model architectures, hyperparameters, and data partition strategies. The chosen datasets were FEMNIST LeCun (1998); Cohen et al. (2017), CelebA Liu et al. (2015), Shakespeare Shakespeare (2014); McMahan et al. (2017), and CIFAR-100 Krizhevsky et al. (2009). An overview of the datasets is given in Table 2. The task of the FEMNIST dataset is classification among 62 classes (10 digits, 26 lowercase letters, 26 uppercase letters) given 28×28 grayscale handwriting images. The task of the CelebA dataset is smile detection, which is modeled as binary classification given 84×84 RGB human face images. Both image classification tasks use CNN models. In contrast, the Shakespeare dataset contains scripts from Shakespeare's work, and the corresponding task is next-character prediction based on sentences of length 80 with an LSTM model. The CIFAR-100 is also a dataset for image classification and contains images of 100 classes.

We applied the model architectures suggested by LEAF for the first three tasks, and a ResNet9 He et al. (2016) for the CIFAR-100 task. The model architectures were provided in the appendix. For the three image classification tasks, we cut the CNN models in the middle so that both client and server part models have similar numbers of layers. For the language processing task, we kept the embeddings and the LSTM cells on the client side while having the projection head on the server side. We applied Adam optimizer for all tasks

Table 2: An overview of datasets.

Dataset	FEMNIST	CelebA	Shakespeare	CIFAR-100
Task	image classification	smile detection	next-char prediction	image classification
Classes	62	2	80	100
Clients	3550	9343	1129	100
Samples per client	226.8 ± 88.9	21.4 ± 7.6	3743.3 ± 6212.3	600.0 ± misc

as it is less sensitive to learning rate compared to SGD, and followed the learning rates suggested by LEAF unless they performed to be too small or large in the experiments. We determined batch sizes according to data distribution among clients and left out clients who had too few samples and could not even fill a full batch. For CIFAR-100, we kept batch size of 64 and conducted grid search for optimal learning rate. An overview of the hyperparameters is provided in Table 8 in the appendix. The choice of optimizers, learning rates, and batch sizes was identical for both clients and server across different SL algorithms in the same task for a fair comparison.

In our benchmarks, we partitioned the data among clients in non-iid ways, which makes the experiments more challenging and closer to real life. For FEMNIST, CelebA, and Shakespeare, we followed the fixed non-iid distributions provided in LEAF. The CIFAR-100 dataset was partitioned using FL-bench, following Dirichlet distribution Hsu et al. (2019) with different α values to emulate different levels of data heterogeneity across clients. Smaller α implies stronger data heterogeneity. The visualizations of data distributions are provided in the appendix. Further we emulated partial participation with a client attendance rate of 5%. This means we randomly sampled 5% of clients in each round, and only these clients could connect with the server. For all tasks, we conducted sample-wise data split Wang et al. (2021), which means for each client, we had a proportion of samples reserved for test, rather than a held-out set of clients who never met the server during training. The train-test split ratio was 90%-10% for all tasks. The main reason for applying sample-wise data split is that for some SL algorithms like PSL and SGLR, there is no model aggregation procedure on the client side, which means the held-out clients receive no updates throughout training, and a test involving these clients is irrelevant.

Our experiments are notably distinct from previous ones, as all four datasets we used contain a great amount of clients, and the data distribution among clients is strongly non-iid, whereas previous experiments often engaged only few clients and little heterogeneity. In other words, our experiments are close to a cross-device Kairouz et al. (2021) case where each client can be an individual user with a personal device like smartphone or tablet and his personal data, whereas existing results were commonly obtained in cross-silo scenarios where large-scale institutions like universities and hospitals act as clients. We believe that the former is more challenging regarding scalability, data heterogeneity, and partial participation. Besides, experiments in cross-device scenarios are more practical in collaborative computing due to the nature of SL.

4.2 Results

We ran our experiments for 600 epochs for the FEMNIST, CelebA, and Shakespeare tasks, and 1000 rounds on the CIFAR-100 dataset. The obtained test metrics are summarized in Table 3, and the graphical visualizations of test loss (cross entropy) were reported in Figure 2. Other metric plots were provided in the appendix. We observed that for FEMNIST, the cycle-version algorithms outperformed all their originals, especially the aggregation-based ones, to a large extent, while the best-performing method was CycleSFL. In detail, CyclePSL, CycleSGLR, and CycleSFL respectively advanced the test accuracies of PSL, SGLR, and SFLV1 from 52.5% to 83.3%, from 48.8% to 82.3%, and from 63.2% to 83.9%. Compared to the aggregation-free method, namely SFLV2, CycleSFL also yielded an accuracy improvement of 1.1%. Another intriguing finding is that CycleSL can also lead to an increase in model robustness and stability, as a decrease in metric standard deviations can be observed for all metrics. For CelebA, since the task is simple (binary), the non-cycle algorithms can already deliver good results. However, CyclePSL, CycleSGLR, and CycleSFL still boosted model performance on average by 4.6%, 3.4%, and 1.3% regarding test accuracy. Moreover, we found out that the improvement was obtained even in the case of overfitting, as can be inferred from the

Table 3: Achieved test loss (cross entropy), accuracy, F1 score, and MCC (Matthews correlation coefficient).

Method	Metric	FEMNIST	CelebA	Shakespeare	CIFAR$_{(iid)}$	CIFAR$_{(\alpha=1.0)}$	CIFAR$_{(\alpha=0.5)}$	CIFAR$_{(\alpha=0.1)}$
PSL	Loss	2.050 ± 0.235	0.648 ± 0.076	3.349 ± 0.133	2.784 ± 0.036	2.638 ± 0.077	2.570 ± 0.083	1.906 ± 0.206
	Accu	0.525 ± 0.046	0.809 ± 0.013	0.167 ± 0.052	0.331 ± 0.026	0.351 ± 0.005	0.368 ± 0.016	0.559 ± 0.051
	F1	0.200 ± 0.037	0.808 ± 0.013	0.006 ± 0.001	0.280 ± 0.016	0.280 ± 0.017	0.280 ± 0.029	0.336 ± 0.041
	MCC	0.506 ± 0.048	0.617 ± 0.026	0.000 ± 0.001	0.325 ± 0.027	0.344 ± 0.005	0.359 ± 0.016	0.542 ± 0.049
SGLR	Loss	2.246 ± 0.248	0.487 ± 0.063	3.336 ± 0.110	2.786 ± 0.043	2.628 ± 0.081	2.581 ± 0.067	1.878 ± 0.187
	Accu	0.488 ± 0.051	0.826 ± 0.022	0.143 ± 0.064	0.319 ± 0.019	0.360 ± 0.034	0.372 ± 0.016	0.572 ± 0.042
	F1	0.168 ± 0.036	0.825 ± 0.023	0.006 ± 0.001	0.262 ± 0.023	0.291 ± 0.028	0.285 ± 0.016	0.353 ± 0.041
	MCC	0.467 ± 0.053	0.653 ± 0.046	0.001 ± 0.001	0.312 ± 0.019	0.354 ± 0.034	0.363 ± 0.016	0.556 ± 0.040
SFLV1	Loss	1.355 ± 0.089	0.217 ± 0.019	1.908 ± 0.101	2.309 ± 0.078	2.253 ± 0.077	2.350 ± 0.090	2.484 ± 0.177
	Accu	0.632 ± 0.021	0.905 ± 0.011	0.454 ± 0.027	0.415 ± 0.032	0.423 ± 0.026	0.416 ± 0.012	0.388 ± 0.062
	F1	0.383 ± 0.028	0.905 ± 0.011	0.147 ± 0.015	0.346 ± 0.027	0.361 ± 0.020	0.339 ± 0.039	0.252 ± 0.039
	MCC	0.618 ± 0.022	0.811 ± 0.021	0.404 ± 0.030	0.410 ± 0.033	0.417 ± 0.026	0.410 ± 0.012	0.380 ± 0.060
SFLV2	Loss	0.509 ± 0.039	0.225 ± 0.023	**1.860 ± 0.128**	2.062 ± 0.132	1.943 ± 0.125	1.963 ± 0.084	1.910 ± 0.077
	Accu	0.828 ± 0.010	0.906 ± 0.009	**0.455 ± 0.032**	0.475 ± 0.041	0.493 ± 0.030	0.480 ± 0.013	0.489 ± 0.039
	F1	0.701 ± 0.016	0.906 ± 0.010	0.156 ± 0.026	0.417 ± 0.041	0.422 ± 0.028	0.409 ± 0.051	0.331 ± 0.036
	MCC	0.822 ± 0.010	0.814 ± 0.017	**0.407 ± 0.035**	0.471 ± 0.041	0.489 ± 0.030	0.474 ± 0.013	0.479 ± 0.042
FedAvg	Loss	1.629 ± 0.071	0.249 ± 0.053	2.337 ± 0.141	2.343 ± 0.068	2.416 ± 0.045	2.551 ± 0.111	2.990 ± 0.166
	Accu	0.567 ± 0.019	0.904 ± 0.017	0.374 ± 0.035	0.411 ± 0.030	0.392 ± 0.021	0.372 ± 0.012	0.296 ± 0.035
	F1	0.293 ± 0.014	0.903 ± 0.017	0.082 ± 0.020	0.342 ± 0.032	0.321 ± 0.012	0.293 ± 0.037	0.169 ± 0.021
	MCC	0.550 ± 0.020	0.812 ± 0.030	0.314 ± 0.042	0.405 ± 0.030	0.386 ± 0.021	0.365 ± 0.012	0.285 ± 0.037
Cycle-PSL	Loss	0.610 ± 0.031	0.687 ± 0.058	5.044 ± 0.187	2.456 ± 0.103	2.328 ± 0.177	2.161 ± 0.121	1.405 ± 0.222
	Accu	0.833 ± 0.016	0.855 ± 0.012	0.108 ± 0.021	0.393 ± 0.016	0.422 ± 0.047	0.465 ± 0.024	**0.650 ± 0.058**
	F1	0.687 ± 0.022	0.854 ± 0.011	0.011 ± 0.001	0.332 ± 0.028	0.350 ± 0.046	0.370 ± 0.030	**0.436 ± 0.033**
	MCC	0.827 ± 0.017	0.710 ± 0.024	0.000 ± 0.002	0.387 ± 0.017	0.416 ± 0.047	0.458 ± 0.025	**0.637 ± 0.057**
Cycle-SGLR	Loss	0.619 ± 0.060	0.548 ± 0.037	5.343 ± 0.347	2.402 ± 0.073	2.290 ± 0.159	2.093 ± 0.051	**1.385 ± 0.219**
	Accu	0.823 ± 0.019	0.860 ± 0.013	0.080 ± 0.014	0.401 ± 0.018	0.427 ± 0.045	0.472 ± 0.029	0.638 ± 0.047
	F1	0.678 ± 0.031	0.860 ± 0.012	0.011 ± 0.001	0.342 ± 0.031	0.348 ± 0.049	0.376 ± 0.031	0.436 ± 0.044
	MCC	0.817 ± 0.019	0.721 ± 0.025	0.000 ± 0.001	0.395 ± 0.019	0.421 ± 0.045	0.465 ± 0.029	0.624 ± 0.046
Cycle-SFL	Loss	**0.489 ± 0.031**	**0.205 ± 0.018**	2.078 ± 0.091	**1.923 ± 0.062**	**1.875 ± 0.085**	**1.850 ± 0.079**	1.825 ± 0.131
	Accu	**0.839 ± 0.008**	**0.918 ± 0.005**	0.437 ± 0.017	**0.491 ± 0.027**	**0.502 ± 0.026**	**0.522 ± 0.031**	0.522 ± 0.029
	F1	**0.710 ± 0.006**	**0.917 ± 0.006**	**0.169 ± 0.016**	**0.433 ± 0.040**	**0.426 ± 0.023**	**0.434 ± 0.042**	0.344 ± 0.041
	MCC	**0.834 ± 0.008**	**0.835 ± 0.011**	0.390 ± 0.018	**0.486 ± 0.027**	**0.497 ± 0.027**	**0.517 ± 0.031**	0.511 ± 0.032

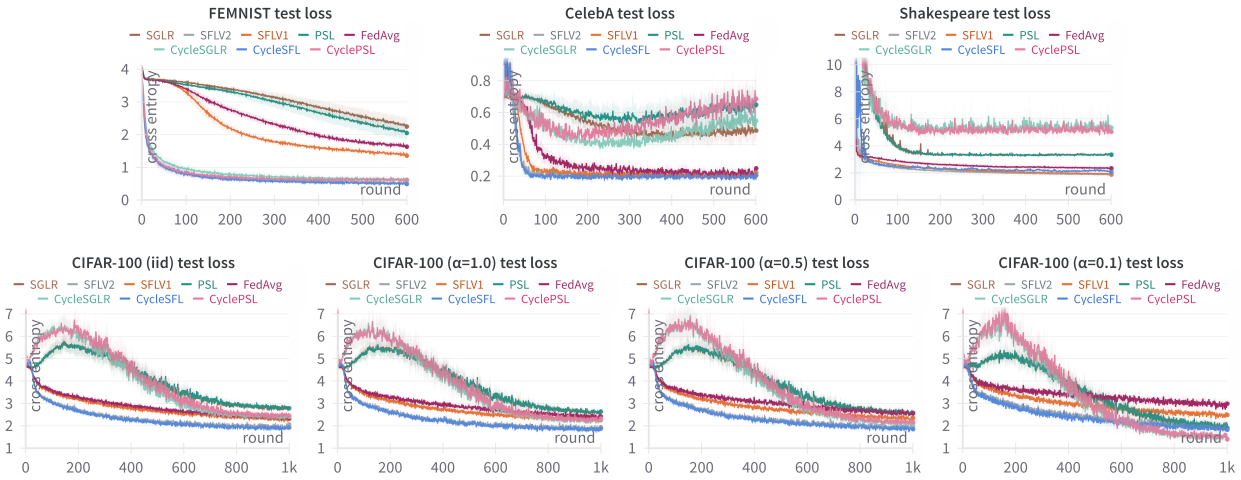


Figure 2: Test loss (cross entropy) for each task.

test loss on CelebA task. When PSL and SGLR and their cycle-versions were overfitted (increase in test loss, but other metrics weren't negatively impacted), the cycle-version methods still yielded lower test loss at earlier rounds. Whilst not being the best-performing method for all metrics on Shakespeare, CycleSFL delivered the highest F1 score and was still on par with SFL with respect to other metrics. For CIFAR-100, we observed that the cycle-version methods generally surpassed their originals across different levels of heterogeneity. Particularly, CycleSFL, which was commonly the best performing method, improved the test accuracy from 47.5% to 49.1% in iid case, from 49.3% to 50.2% when $\alpha = 1.0$, from 48.0% to 52.2% when

Table 4: Minimal epochs required to reach certain test accuracy for each task (45% for FEMNIST, 75% for CelebA, 35% for Shakespeare, 30% for CIFAR-100). Smaller is better.

Method	FEMNIST	CelebA	Shakespr	$\text{CIFAR}_{(iid)}$	$\text{CIFAR}_{(\alpha=1.0)}$	$\text{CIFAR}_{(\alpha=0.5)}$	$\text{CIFAR}_{(\alpha=0.1)}$
PSL	459	255	> 600	> 1000	> 1000	> 1000	476
SGLR	517	244	> 600	> 1000	> 1000	> 1000	476
SFLV1	184	39	154	293	253	289	342
SFLV2	8	31	105	131	118	149	**162**
FedAvg	287	65	386	335	337	488	> 1000
CyclePSL	8	94	> 600	584	552	526	422
CycleSGLR	10	94	> 600	581	534	532	395
CycleSFL	**6**	**29**	**98**	**116**	**107**	**132**	**162**

$\alpha = 0.5$, and from 48.9% to 52.2% when $\alpha = 0.1$, respectively, compared to the second best method SFLV2. Another noticeable finding is that with the increase of data heterogeneity (α decreases), algorithms that do not require model aggregation on the client side like PSL and SGLR began to overtake the aggregation-based methods, which could be attributed to model personalization on the client side Tan et al. (2022). Still, the cycle-verion methods, namely CyclePSL and CycleSGLR, maintained their dominance over PSL and SGLR vastly, with an increase in test accuracy from 55.9% to 65.0% and from 57.2% to 63.8% under extreme data heterogeneity ($\alpha = 0.1$).

CycleSL is also promising in terms of convergence rate. We measured the convergence speed of each algorithm by recording the first time that their test accuracies surpassed certain thresholds (45% for FEMNIST, 75% for CelebA, 35% for Shakespeare, 30% for CIFAR-100) in Table 4. We could learn from the tables and figures that the cycle-version methods commonly converged at (much) earlier stages, especially compared to their aggregation-based originals, unless the original methods failed to converge, while CycleSFL was overall the fastest algorithm in this benchmark across all tasks. For instance, on the FEMNIST dataset, PSL, SGLR, and SFLV1 only started to make noticeable progress in test loss after 100 epochs, whereas their cycle-versions CyclePSL, CycleSGLR, and CycleSFL almost reached their convergence around 100th round. Similar phenomena can be observed for other tasks as well. Especially for the CelebA task, although PSL and SGLR started to overfit after roughly 300 and 400 rounds, CyclePSL and CycleSGLR still delivered lower test loss at earlier time points.

4.3 Ablation study

4.3.1 Impact of cut layer

In contrast to FL, in SL, the choice of cut layer plays an key role for many factors such as model performance, computation overhead, transmission cost, and privacy risk. In the following, we mainly concentrated on the impact of cut layer on model performance, and carried out ablation study on the CIFAR-100 dataset using the ResNet9 model with different levels of data heterogeneity. To simplify the experiments, we employed solely one algorithm, namely CycleSFL, and explored the influence of block-wise cut point rather than layer-wise. The ResNet9 model contains 4 convolutional blocks, 2 residual blocks, and 1 projection head. Therefore there are totally 6 different possible cut positions. We then cut the model at each possible point and recorded the achieved test accuracy in Table 5. As can be inferred from the table, a shallower cut point can lead to better model performance. We attributed this to that since in CycleSL there is only one model on the server side, the impact of data heterogeneity and client drift mostly resides in client models. In such a case, a reduced model complexity on the client side, i.e., a shallower cut layer, would help improve convergence. However, the choice of cut layer is a complex bargain game between clients and server involving many other factors like privacy and transmission overhead. For example, a shallower cut layer may lead to data leak, as input data can be easier reconstructed due to stronger correlation between input data and activations. For detailed analysis in this regard, we direct readers to relevant works Yan et al. (2022); Wu et al. (2023); Kim et al. (2023).

Table 5: Impact of cut layer on CycleSFL test accuracy for the CIFAR-100 task.

Cut	*iid*	$\alpha = 1.0$	$\alpha = 0.5$	$\alpha = 0.1$
1	0.531±0.039	0.558±0.026	0.542±0.008	0.547±0.037
2	0.517±0.027	0.533±0.043	0.516±0.023	0.531±0.050
3	0.487±0.035	0.511±0.020	0.500±0.027	0.511±0.040
4	0.475±0.017	0.479±0.019	0.479±0.030	0.496±0.019
5	0.443±0.020	0.466±0.033	0.460±0.017	0.468±0.029
6	0.427±0.030	0.414±0.027	0.412±0.013	0.376±0.048

Table 6: Impact of server epoch on CycleSFL test accuracy for the CIFAR-100 task.

Epoch	*iid*	$\alpha = 1.0$	$\alpha = 0.5$	$\alpha = 0.1$
1	0.487±0.013	0.518±0.025	0.502±0.021	0.518±0.039
2	0.503±0.014	0.520±0.010	0.522±0.024	0.587±0.041
4	0.464±0.013	0.497±0.030	0.472±0.038	0.587±0.033
8	0.477±0.020	0.497±0.029	0.498±0.034	0.622±0.053

4.3.2 Impact of server epoch

In the original FL work McMahan et al. (2017), the authors suggested a simple but effective way to boost convergence, namely by allowing clients to train for multiple epochs before aggregation. In CycleSL, since the training on the server side is modelled as a standalone task, similar strategy can be applied on the server side, namely allowing the server to train its model for multiple rounds before computing gradients for clients. To this end, we investigated the impact of server epoch on model performance on the CIFAR-100 dataset with different levels of data distribution heterogeneity among clients. For simplicity we merely benchmarked one method, namely CycleSFL, with the number of server training epochs scaling from 1 to 8. The obtained test accuracy was listed in Table 6. From the table we can observe that when data heterogeneity is not drastic (iid, $\alpha = 1.0$, $\alpha = 0.5$), the increase of server training pass from 1 to 2 generally led to better model performance. However, further increment of server round resulted in a decrease of test accuracy. In contrast, under extreme distribution heterogeneity ($\alpha = 0.1$), the increase of server epoch up to 8 can consistently improve model performance. We believed that the choice of server training round in CycleSL is a trade-off between model personalization Tan et al. (2022) and regularization, as a more trained server part model would yield lower objective value and hence smaller gradient steps for clients, which in turn motivates client models to converge to similar local optima but reduces local model personalization. In general, the increment of server training epoch is associated with better model performance when both the number of epochs and data heterogeneity are not high.

5 Discussion

5.1 Application and future work

As CycleSL can be easily combined with other scalable SL methods, its applications can be valuable in practice. Particularly, the choice of foundation algorithm could be determined according to the specific case. For instance, when client side aggregation is risky due to privacy concerns or model personalization is wished, CyclePSL and CycleSGLR can be employed. In contrast, if a performant model and faster convergence are desired, CycleSFL is a good option. Given its low burden and high scalability, CycleSL can be particularly beneficial in cross-device cases where the number of participating clients is large and the computing resource per client is generally limited, such as in edge computing scenarios. Moreover, since the server part training is an isolated task in CycleSL, an independent and flexible setting for hyperparameters such as model split, regularization method, and aggregation frequency Lin et al. (2024a) could be applied to further advance model performance and convergence rate. Due to its robustness and implicit regularization effect, CycleSL is more tolerant of hyperparameters such as learning rates and batch sizes, which can be costly to tune,

especially in privacy-sensitive scenarios. The feature resampling strategy of CycleSL lays the foundation for future works like ensemble learning and knowledge distillation in SL, which has already become popular in the context of FL Lin et al. (2020); Attota et al. (2021); Wu et al. (2024). From the perspective of optimization theory, CycleSL can be regarded as coordinate descent on function composition, whilst the research for its convergence property like Nesterov (2013); Richtárik & Takáč (2014); Zeng et al. (2019) is in an early stage. Further study is needed in this regard. The impact of model cut point on specific cycle-version method also awaits further investigation. In addition, since smashed data from clients is regarded as input samples for a standalone task in CycleSL, client selection or reinforcement based on client features could be a further research direction. To this end, techniques like sample-wise or client-wise attention mechanisms Cheng et al. (2021) for smashed data may be considered. It should also be noticed that for simplicity we restrict our discussion to SL with label sharing Gupta & Raskar (2018); Vepakomma et al. (2018) in this work, meaning that there is only one split point and clients send both extracted features and labels to server. The methods can be easily extended to SL without label sharing with two or even more model cut points on demand.

5.2 Limitations

The integration of CycleSL can notably reduce the memory and computation burden on the server side over the aggregation-based algorithms. One drawback of CycleSL is that its latency and computation burden is larger than SFLV2, as SFLV2 only conducts forward and backward propagation once on the server side, whereas the smashed data is fed into the server part model twice in CycleSFL. However, the memory cost of CycleSFL is lower compared to SFLV2, since CycleSFL only computes gradients with respect to one set of parameters each time while the other set is frozen. And the latency and computation can be lightened with proper choice of hyperparameters such as batch size for the second-stage task. Another limitation of CycleSL is that the combination of CycleSL and other scalable algorithms may inherit the problems of those methods themselves. For instance, we learnd from the test loss on the CelebA dataset that PSL and SGLR began to overfit after roughly 300 and 400 iterations, respectively. Although the integration of CycleSL brought a decrease in test loss at early rounds, CyclePSL and CycleSGLR did not refrain from overfitting. Similarly, if the original algorithm fails to converge, the incorporation of CycleSL may not converge as well, as can be inferred from the results of CyclePSL and CycleSGLR on the Shakespeare dataset. Further, an imbalanced model split could lead to suboptimal performance of CycleSL, as CycleSFL, with only a single linear layer on the server side, cannot outperform SFL in the Shakespeare task.

5.3 Privacy concern

The integration of CycleSL into other scalable SL methods does not require any additional data or model transfer. Therefore, the privacy claims of the original algorithms still apply, and the former privacy-enhancing mechanisms, like differential privacy and k-anonymity, would still work. Furthermore, considering its aggregation-free property, we believe CycleSL is more robust against client-level noise injection than the aggregation-based methods and has a good potential to improve the privacy-utility trade-off, as handling noisy models is, in general, a harder task than handling noisy samples. Moreover, since the smashed data is resampled on the server side and not client-binding anymore, the noise in the hidden representations could be compensated over features from different clients. Random resampling can also contribute to model robustness and stability. The impact of malicious participants could also be reduced with such a procedure.

6 Conclusion

In this work, we presented a novel aggregation-free scalable SL algorithm called CycleSL, which models the server part training as a standalone higher-level task and benefits from coordinate descent. Particularly, CycleSL applies resampling to smashed data to counteract client drift in the higher-level task, and updates the server and clients in cyclical turns, as inspired by coordinate descent. CycleSL can be combined with other scalable SL methods to boost model performance and reduce resource consumption. By integrating CycleSL into existing methods, including PSL, SGLR, and SFL, we introduced CyclePSL, CycleSGLR, and CycleSFL accordingly. Our results show that CycleSL can notably improve model performance and convergence.

References

Ali Abedi and Shehroz S Khan. Fedsl: Federated split learning on distributed sequential data in recurrent neural networks. *Multimedia Tools and Applications*, 2023.

Dinesh Chowdary Attota, Viraaji Mothukuri, Reza M Parizi, and Seyedamin Pouriyeh. An ensemble multi-view federated learning intrusion detection for iot. *IEEE Access*, 9:117734–117745, 2021.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19, 2006.

Sebastian Caldas, Sai Meher Karthik Duddu, Peter Wu, Tian Li, Jakub Konečnỳ, H Brendan McMahan, Virginia Smith, and Ameet Talwalkar. Leaf: A benchmark for federated settings. *arXiv:1812.01097*, 2018.

Miguel Carreira-Perpinan and Weiran Wang. Distributed optimization of deeply nested systems. In *Artificial Intelligence and Statistics*, pp. 10–19. PMLR, 2014.

Zachary Charles and Jakub Konečnỳ. Convergence and accuracy trade-offs in federated learning and meta-learning. In *International Conference on Artificial Intelligence and Statistics*, pp. 2575–2583. PMLR, 2021.

Qishang Cheng, Hongliang Li, Qingbo Wu, and King Ngi Ngan. Baˆ 2m: A batch aware attention module for image classification. *arXiv preprint arXiv:2103.15099*, 2021.

Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 international joint conference on neural networks (IJCNN)*, pp. 2921–2926. IEEE, 2017.

Ingrid Daubechies, Michel Defrise, and Christine De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. *Communications on Pure and Applied Mathematics: A Journal Issued by the Courant Institute of Mathematical Sciences*, 57(11):1413–1457, 2004.

Wenjiang J Fu. Penalized regressions: the bridge versus the lasso. *Journal of computational and graphical statistics*, 7(3):397–416, 1998.

Tobias Glasmachers. Limits of end-to-end learning. In *Asian conference on machine learning*, pp. 17–32. PMLR, 2017.

Otkrist Gupta and Ramesh Raskar. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116:1–8, 2018.

Dong-Jun Han, Hasnain Irshad Bhatti, Jungmoon Lee, and Jaekyun Moon. Accelerating federated learning with split learning on locally generated losses. In *ICML 2021 workshop on federated learning for user privacy and data confidentiality. ICML Board*, 2021.

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

Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

Tzu-Ming Harry Hsu, Hang Qi, and Matthew Brown. Measuring the effects of non-identical data distribution for federated visual classification. *arXiv preprint arXiv:1909.06335*, 2019.

Joohyung Jeon and Joongheon Kim. Privacy-sensitive parallel split learning. In *2020 International Conference on Information Networking (ICOIN)*, pp. 7–9. IEEE, 2020.

Praveen Joshi, Chandra Thapa, Seyit Camtepe, Mohammed Hasanuzzamana, Ted Scully, and Haithem Afli. Splitfed learning without client-side synchronization: Analyzing client-side split network portion size to overall performance. *arXiv preprint arXiv:2109.09246*, 2021.

Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and trends® in machine learning*, 14(1–2):1–210, 2021.

Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. Scaffold: Stochastic controlled averaging for federated learning. In *International conference on machine learning*, pp. 5132–5143. PMLR, 2020.

Minsu Kim, Alexander DeRieux, and Walid Saad. A bargaining game for personalized, energy efficient split learning over wireless networks. In *2023 IEEE Wireless Communications and Networking Conference (WCNC)*, pp. 1–6. IEEE, 2023.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images, 2009.

Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. Ensemble distillation for robust model fusion in federated learning. *Advances in neural information processing systems*, 33:2351–2363, 2020.

Zheng Lin, Guanqiao Qu, Wei Wei, Xianhao Chen, and Kin K Leung. Adaptsfl: Adaptive split federated learning in resource-constrained edge networks. *arXiv:2403.13101*, 2024a.

Zheng Lin, Guangyu Zhu, Yiqin Deng, Xianhao Chen, Yue Gao, Kaibin Huang, and Yuguang Fang. Efficient parallel split learning over resource-constrained wireless edge networks. *IEEE Transactions on Mobile Computing*, 2024b.

Ziwei Liu, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Deep learning face attributes in the wild. In *Proceedings of the IEEE international conference on computer vision*, 2015.

David G Luenberger, Yinyu Ye, et al. *Linear and nonlinear programming*, volume 2. Springer, 1984.

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

Kensuke Nakamura, Stefano Soatto, and Byung-Woo Hong. Block-cyclic stochastic coordinate descent for deep neural networks. *Neural Networks*, 139:348–357, 2021.

Yu Nesterov. Gradient methods for minimizing composite functions. *Mathematical programming*, 140(1): 125–161, 2013.

Shraman Pal, Mansi Uniyal, Jihong Park, Praneeth Vepakomma, Ramesh Raskar, Mehdi Bennis, Moongu Jeon, and Jinho Choi. Server-side local gradient averaging and learning rate acceleration for scalable split learning. *arXiv preprint arXiv:2112.05929*, 2021.

Laura Palagi and Ruggiero Seccia. Block layer decomposition schemes for training deep neural networks. *Journal of Global Optimization*, 77(1):97–124, 2020.

John Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. Technical Report MSR-TR-98-14, Microsoft, April 1998. URL `https://www.microsoft.com/en-us/research/publication/sequential-minimal-optimization-a-fast-algorithm-for-training-support-vector-machines/`.

Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

Peter Richtárik and Martin Takáč. Iteration complexity of randomized block-coordinate descent methods for minimizing a composite function. *Mathematical Programming*, 144(1):1–38, 2014.

William Shakespeare. *The complete works of William Shakespeare*. Race Point Publishing, 2014.

Elif Ustundag Soykan, Leyli Karacay, Ferhat Karakoc, and Emrah Tomur. A survey and guideline on privacy enhancing technologies for collaborative machine learning. *IEEE Access*, 10:97495–97519, 2022.

Alysa Ziying Tan, Han Yu, Lizhen Cui, and Qiang Yang. Towards personalized federated learning. *IEEE transactions on neural networks and learning systems*, 34(12), 2022.

Jiahao Tan and Xinpeng Wang. FL-bench: A federated learning benchmark for solving image classification tasks, 2023. URL `https://github.com/KarhouTam/FL-bench`.

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

Valeria Turina, Zongshun Zhang, Flavio Esposito, and Ibrahim Matta. Federated or split? a performance and privacy analysis of hybrid split and federated learning architectures. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

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

Jianyu Wang, Zachary Charles, Zheng Xu, Gauri Joshi, H Brendan McMahan, Maruan Al-Shedivat, Galen Andrew, Salman Avestimehr, Katharine Daly, Deepesh Data, et al. A field guide to federated optimization. *arXiv preprint arXiv:2107.06917*, 2021.

Mengdi Wang, Anna Bodonhelyi, Efe Bozkir, and Enkelejda Kasneci. Turbosvm-fl: Boosting federated learning through svm aggregation for lazy clients. *arXiv preprint arXiv:2401.12012*, 2024.

Stephen J Wright. Coordinate descent algorithms. *Mathematical programming*, 151(1):3–34, 2015.

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

Xing Wu, Jie Pei, Xian-Hua Han, Yen-Wei Chen, Junfeng Yao, Yang Liu, Quan Qian, and Yike Guo. Fedel: Federated ensemble learning for non-iid data. *Expert Systems with Applications*, 237:121390, 2024.

Jia Yan, Suzhi Bi, and Ying-Jun Angela Zhang. Optimal model placement and online model splitting for device-edge co-inference. *IEEE Transactions on Wireless Communications*, 21(10):8354–8367, 2022.

Jinshan Zeng, Tim Tsz-Kit Lau, Shaobo Lin, and Yuan Yao. Global convergence of block coordinate descent in deep learning. In *International conference on machine learning*, pp. 7313–7323. PMLR, 2019.

Ziming Zhang and Matthew Brand. Convergent block coordinate descent for training tikhonov regularized deep neural networks. *Advances in Neural Information Processing Systems*, 30, 2017.

Zongshun Zhang, Andrea Pinto, Valeria Turina, Flavio Esposito, and Ibrahim Matta. Privacy and efficiency of communications in federated split learning. *IEEE Transactions on Big Data*, 2023.

Peng Zhao, Yun Yang, and Qiao-Chu He. High-dimensional linear regression via implicit regularization. *Biometrika*, 109(4):1033–1046, 2022.

Yunhong Zhou, Dennis Wilkinson, Robert Schreiber, and Rong Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Algorithmic Aspects in Information and Management: 4th International Conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings 4*, pp. 337–348. Springer, 2008.

Appendix

A Implementation

We implemented in total eight SL/FL algorithms, including PSL, SFLV1, SFV2, SGLR, FedAvg, CyclePSL, CycleSFL, and CycleSGLR. All methods were implemented with PyTorch. An overview of our experiment environment is given in Table 7. Additionally we provided the code for sequential SL (SSL) and its cycle-version (CycleSSL). Since they are not scalable methods, we excluded them from benchmark. Our implementation can be found on `https://github.com/AnonymWriter/CycleSL` (This repository is anonymized for review). By following the instructions we provided, our experiment results should be completely reproducible up to numerical error.

B Randomness

To ensure the robustness and reliability of results, we replicated the experiments for five times, with each time being initialized with a different and unique random seed from $\{0, 1, 2, 3, 4\}$. The random seed was fed to all libraries which can potentially be influenced by randomness in the beginning, including *numpy.random.seed*, *torch.manual_seed*, and *random.seed*. We reported all our metrics in form of mean $\pm$ std over the five seeds.

C Environment

The benchmarks were conducted on a computer with AMD EPYC 7763 and NVIDIA A100 80GB PCIe. All SL algorithms were implemented in PyTorch. An overview of the hardware and software of our environment is given in Table 7. It should be noticed that although we conducted the experiments on a powerful machine, we have tried to optimize our code so that it can be run on a normal PC as well, even without dedicated GPU.

Table 7: Experiment environment by April 29th, 2025.

Hardware	**Specification**
CPU	AMD EPYC 7763 64-Core
GPU	NVIDIA A100 80GB PCIe×4
Memory	1TB

Software	**Version**
OS	Ubuntu 22.04.5 LTS
Python	3.12.7 by Anaconda
PyTorch	2.5.1 for CUDA 12.4
Scikit-Learn	1.6.0
WandB	0.19.1

D Hyperparameters

For FEMNIST, CelebA, and Shakespeare, we followed the hyperparameters suggested by LEAF, especially the learning rates, unless they performed to be too small or too large in the experiment. For CIFAR-100, we kept batch size of 64 and conducted grid search for learning rate in range of $\{1e{-}5, 1e{-}4, 1e{-}3, 1e{-}2, 1e{-}1\}$ for each algorithm. The overall best performing learning rate was $1e-4$. An overview of hyperparameters is given in Table 8.

E Model architecture

For FEMNIST, CelebA, and Shakespeare, we followed the model structures as suggested by LEAF (`https://github.com/TalwalkarLab/leaf/tree/master/models`), which are CNN, CNN, and LSTM, respectively. For the two CNN models, we cut them in the middle such that client and server parts have similar numbers of layers. For the LSTM model, we kept the embeddings and recurrent cells on the client

Table 8: Details of hyperparameters (consistent for clients and server across SL methods unless specifically mentioned).

Dataset	**FEMNIST**	**CelebA**	**Shakespeare**	**CIAFR-100**
Batch Size	32	16	32	64
Optimizer	Adam	Adam	Adam	Adam
Learning Rate	$3e-4$	$1e-2$	$3e-2$	$1e-4$

while the projection head on the server. The model architectures, sources, and cut points are summarized in Tables 9–11 respectively.

Table 9: CNN architecture for the FEMNIST task. Source: `https://github.com/TalwalkarLab/leaf/blob/master/models/femnist/cnn.py`.

Layer	**Specification**
Input	shape $1 \times 28 \times 28$
Conv2d	kernel size 5, in/out channel 1/32, same padding
ReLU	-
MaxPooling	kernel size 2, stride 2
Conv2d	kernel size 5, in/out channel 32/64, same padding
ReLU	-
MaxPooling	kernel size 2, stride 2
Cut Layer	**client/server cut point**
Flatten	-
Linear	in/out dimension 3136/2048
ReLU	-
Linear	in/out dimension 2048/62

Table 10: CNN architecture for the CelebA task. Source: `https://github.com/TalwalkarLab/leaf/blob/master/models/celeba/cnn.py`.

Layer	**Specification**
Input	shape $3 \times 84 \times 84$
Conv2d	kernel size 3, in/out channel 3/32, same padding
BatchNorm2d	-
MaxPolling	kernel size 2, stride 2
ReLU	-
Conv2d	kernel size 3, in/out channel 32/32, same padding
BatchNorm2d	-
MaxPolling	kernel size 2, stride 2
ReLU	-
Cut Layer	**client/server cut point**
Conv2d	kernel size 3, in/out channel 32/32, same padding
BatchNorm2d	-
MaxPolling	kernel size 2, stride 2
ReLU	-
Conv2d	kernel size 3, in/out channel 32/32, same padding
BatchNorm2d	-
MaxPolling	kernel size 2, stride 2
ReLU	-
Flatten	-
Linear	in/out dimension 800/2

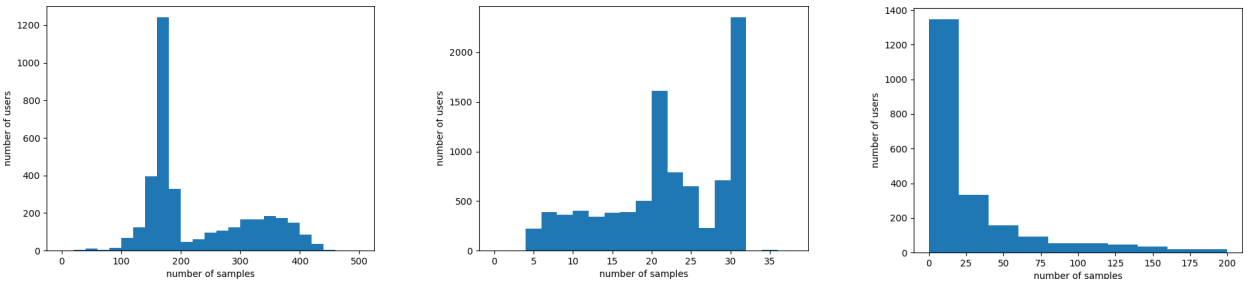
Table 11: LSTM architecture for the Shakespeare task. Source: `https://github.com/TalwalkarLab/leaf/blob/master/models/shakespeare/stacked_lstm.py`.

Layer	**Specification**
Embedding	number of embeddings 80, dimension 8
LSTM	3n/hidden dimension 8/256, hidden layers 2
Cut Layer	**client/server cut point**
Linear	in/out dimension 256/80

For the CIFAR-100 task, we adopted an ResNet9 network He et al. (2016). Our implementation followed `https://www.kaggle.com/code/kmldas/cifar10-resnet-90-accuracy-less-than-5-min?scriptVersionId=38462746&cellId=28`). ResNet9 contains four convolutional blocks, two residual blocks, and a projection head. To balance the number of layers, we kept two convolutional blocks and one residual block on the client side, while the rest and the projection head on the server side. Implementation details can be found in `https://github.com/AnonymWriter/CycleSL/blob/main/models.py`. In the ablation study, we further investigated the influence of cut point on CycleSL. The corresponding results were discussed in subsection 4.3.1.

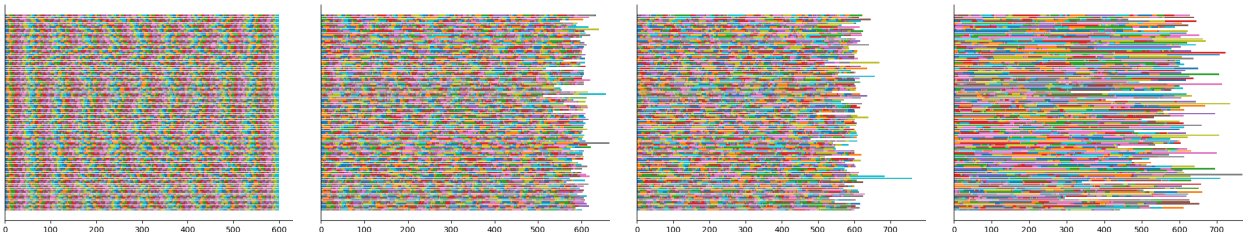
F Data distribution

The histograms of number of samples per client of the FEMNIST, CelebA, and Shakespeare datasets are given in Figure 3. Particularly, the CIFAR-100 dataset was partitioned with Dirichlet distribution using different α values to emulate different levels of data heterogeneity across clients (smaller α implies stronger data heterogeneity). The partition was done via FL-bench Tan & Wang (2023). The impact of α on label distribution can be observed in Figure 4.



(a) FEMNIST (b) CelebA (c) Shakespeare

Figure 3: Histograms of number of samples per user for the FEMNIST, CelebA, and Shakespeare datasets.



(a) iid (b) $\alpha = 1.0$ (c) $\alpha = 0.5$ (d) $\alpha = 0.1$

Figure 4: Label distributions among clients in CIFAR-100 (smaller α implies stronger data heterogeneity).

G Additional results

G.1 Metric plots

The test metrics, inlcluding loss (cross entropy), accuracy, F1 score, and MCC (Matthews correlation coefficient), were plotted in Figures 5–11, respectively. It should be noticed that although some methods like PSL and SGLR overfitted for CelebA (increase in test loss), metrics like accuracy and F1 score were not negatively impacted. Hence we still reported metrics around 600th epoch.

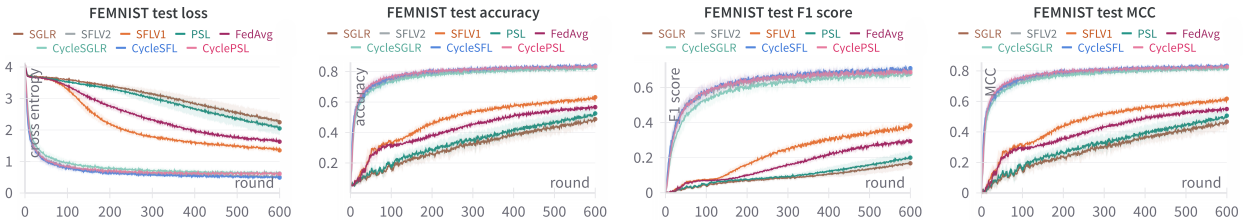


Figure 5: Test metrics for the FEMNIST task.

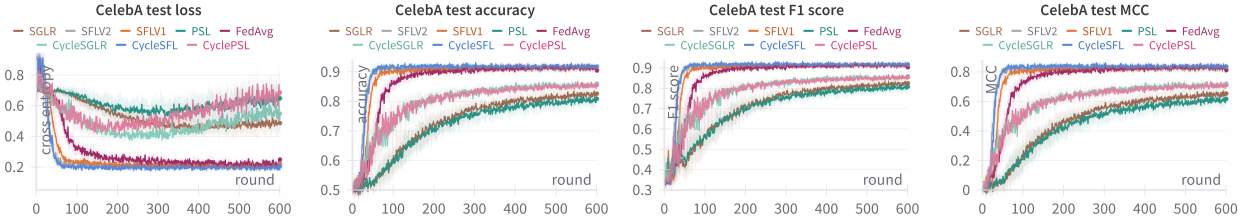


Figure 6: Test metrics for the CelebA task.

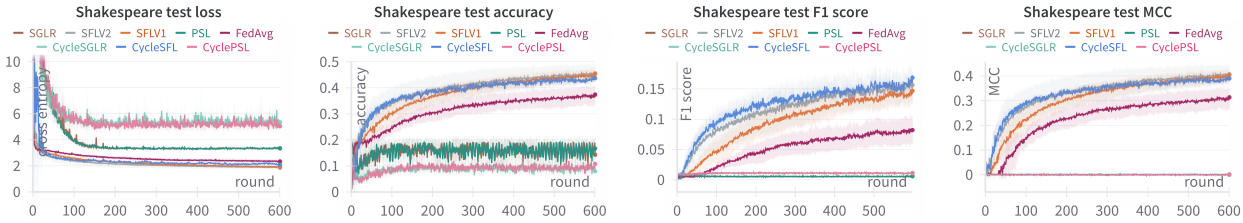


Figure 7: Test metrics for the Shakespeare task.

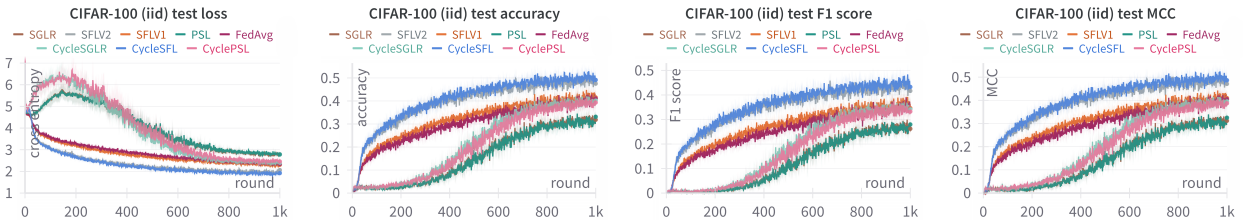


Figure 8: Test metrics for the CIFAR-100 task (iid).

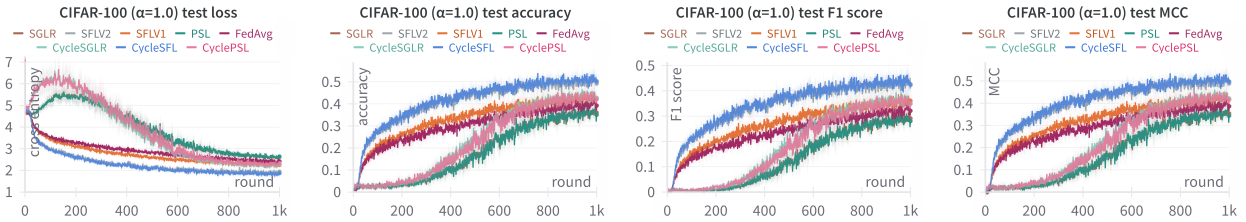


Figure 9: Test metrics for the CIFAR-100 task ($\alpha = 1.0$).

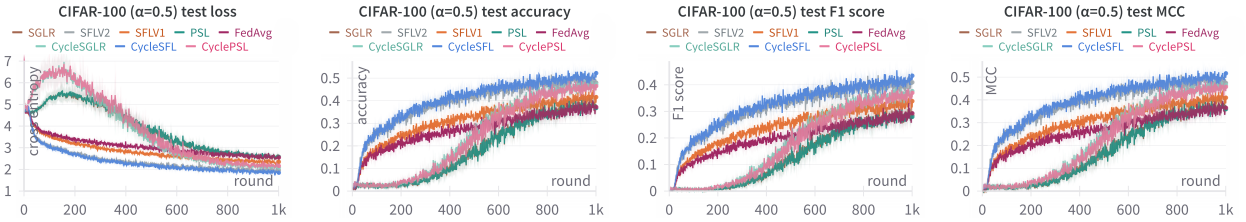


Figure 10: Test metrics for the CIFAR-100 task ($\alpha = 0.5$).

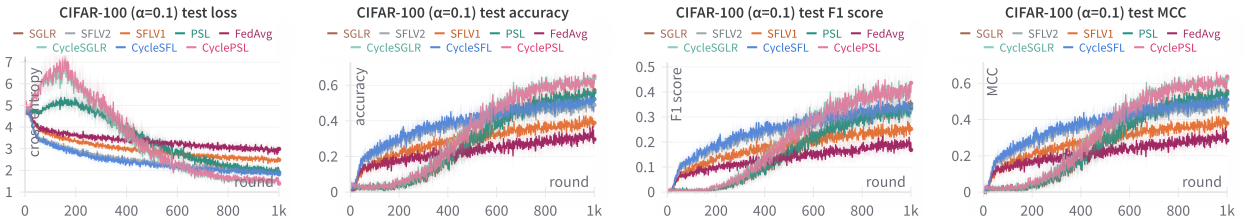
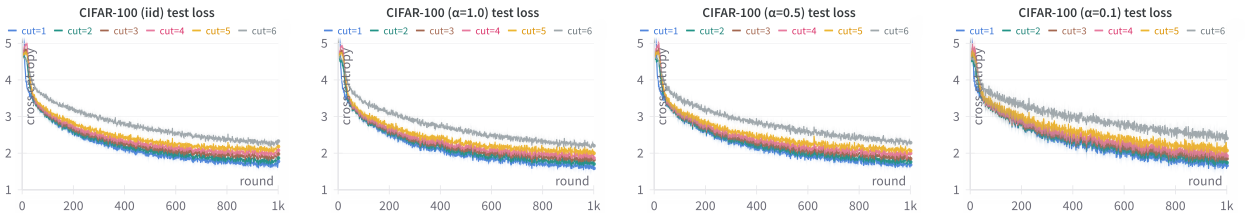


Figure 11: Test metrics for the CIFAR-100 task ($\alpha = 0.1$).

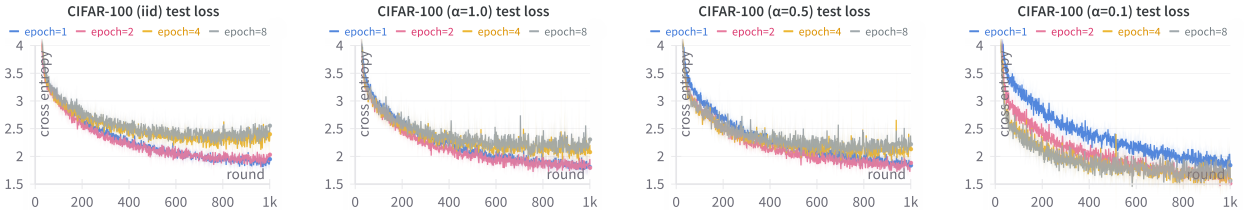
G.2 Ablation study - impact of cut layer and server round

The impact of cut layer (block-wise) on CycleSFL test loss on the CIFAR-100 dataset is plotted in Figure 12. And the influence of server epoch on CycleSFL test loss on the CIFAR-100 dataset is visualized in Figure 13.



(a) iid (b) $\alpha = 1.0$ (c) $\alpha = 0.5$ (d) $\alpha = 0.1$

Figure 12: Impact of cut layer on CycleSFL test loss on CIFAR-100.



(a) iid (b) $\alpha = 1.0$ (c) $\alpha = 0.5$ (d) $\alpha = 0.1$

Figure 13: Impact of server epoch on CycleSFL test loss on CIFAR-100.