Towards a Unified Framework for Split Learning

Boris Radovič
KAUST, University of Ljubljana

Marco Canini
KAUST

Samuel Horváth
MBZUAI

Veljko Pejović
University of Ljubljana

Praneeth Vepakomma
MBZUAI, MIT

Abstract

Split Learning (SL) is a principled approach for training models on data distributed across multiple devices without sharing training data. While SL emerged as an alternative to federated learning to reduce the compute burden on devices, it also enables to redistribute work across nodes. Despite its potential, there is no unified framework for implementing and deploying SL algorithms, leaving several research questions underexplored. To address this gap, we introduce SplitBud, a versatile framework to implement virtually any SL algorithm. By supporting various variants of SL, SplitBud facilitates research and development in the field. In this paper, we demonstrate its flexibility by implementing and evaluating multiple SL algorithms, and we discuss future directions for the field.

CCS Concepts

• **Computing methodologies → Machine learning**; *Distributed algorithms.*

Keywords

Distributed Collaborative Machine Learning, Split Learning

ACM Reference Format:
Boris Radovič, Marco Canini, Samuel Horváth, Veljko Pejović, and Praneeth Vepakomma. 2025. Towards a Unified Framework for Split Learning. In *The 5th Workshop on Machine Learning and Systems (EuroMLSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3721146.3721936

1 Introduction

The rapid growth of machine learning models in both size and complexity demands an ever-increasing amount of data to improve performance. However, public datasets are finite, and projections indicate that the supply of publicly available human text data may be exhausted between 2026 and 2032 [41]. Leveraging private data, which is often distributed across devices (such as smartphones) and organizations (such as hospitals), is becoming crucial to sustain further advancements in machine learning. As collecting all distributed data on a cluster for centralized training remains infeasible due to privacy concerns and legal constraints [26], the field of Distributed Collaborative Machine Learning (DCML) has emerged, enabling models to be trained on private data without exposing sensitive information [33].

Within DCML, Federated Learning (FL) [22, 26, 34] was initially proposed to enable clients to collaboratively train a model with the assistance of a server. In FL, clients create a model through multiple training iterations. At the beginning of each iteration, they receive training parameters from the server.[1] Then, they locally train a model using their private data and return the training outcome (e.g., updated model weights) to the server. The server aggregates these updates to produce the training parameters for the next iteration.

A major limitation of FL is that clients must fully train a model locally. This is problematic because many real-world devices operate under resource constraints [20], making it impractical to train large models on them. While various FL algorithms allow clients to train models proportional to their compute capacity [7, 18, 19, 23], these approaches often lead to lower model performance than if all clients jointly trained a large model [33]. As state-of-the-art model architectures continue to grow in size [42], this challenge is becoming even more pronounced. For instance, to date, efforts for training large language models with FL have been largely restricted to the cross-silo setting, where only a small number of computationally powerful clients participate [8, 10]. Thus, training large models on edge devices while preserving privacy remains an open problem.

To address these limitations, Split Learning (SL) has emerged as a promising alternative. SL extends FL by partitioning the

ACM ISBN 979-8-4007-1538-9/25/03
https://doi.org/10.1145/3721146.3721936

[1] Clients typically receive the current model weights [26, 34], but the server may also send other data, such as logits [19, 23], prototypes [36], or auxiliary model weights [43]. Additionally, the server may provide hyperparameters such as learning rate or batch size. Throughout this paper, we refer to all data and configurations sent to clients as *training parameters.*

model's computational graph between clients and a compute-powerful server. Typically, clients train the initial layers of the model and offload intermediate representations to the server, which processes the remaining layers [15, 40]. This offload of computation reduces both the memory and computational burden on the clients, enabling them to train larger models than they could independently.

Given these advantages, we argue that SL is essential for training large models on private, decentralized data. However, most existing SL algorithms have been evaluated only in simulated environments, where clients are emulated as software components rather than deployed on physical devices [16, 28, 39]. Even when real devices are used, researchers often develop ad-hoc implementations of their algorithms [11, 12]. To the best of our knowledge, *no existing framework allows researchers and practitioners to easily design, evaluate, and deploy SL algorithms.* A standardized framework would allow researchers to focus on algorithm development rather than infrastructure, while also improving reproducibility.

The absence of such a framework has also contributed to several underexplored aspects of SL training. For example, different SL algorithms vary in how they manage server-side models. At one extreme, all clients may train a single shared server model concurrently; at the other, each client may train its own server-side model, with aggregation occurring only after all clients complete training [39]. Optimizing the number of server models is challenging, as it affects both convergence behavior and system efficiency metrics such as GPU utilization, training time, and throughput. A standardized framework would enable fair comparisons between different strategies.

To fill this gap, we introduce SplitBud, the first general framework for split training and inference. SplitBud supports a wide range of SL use cases, enabling researchers and practitioners to implement SL algorithms with minimal code, as we demonstrate in this paper. It also provides a simple abstraction of the key components, facilitating a clearer understanding of SL algorithms. While we focus on split learning, SplitBud is task-agnostic and can also support inference-time applications, such as early exit strategies [29, 38].

In this paper, we discuss the design and implementation of SplitBud. We then demonstrate through examples how existing algorithms can be implemented in the framework, and showcase the benefits of training models with SL using the framework. We also discuss future directions for SL.

In summary, the contributions of this paper are:

- We propose SplitBud, a new flexible SL framework.
- We evaluate the framework in different settings highlighting the benefits of training a model with SL over FL.

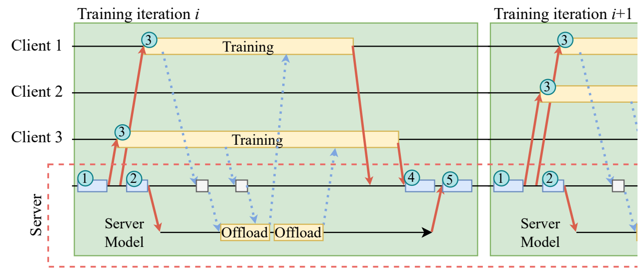

Figure 1: Split Learning Protocol. Solid red lines represent the exchange of training parameters and updates. Dotted blue lines indicate computational graph offloading (for simplicity, only one batch per client is shown). Upon receiving a request, the server determines the server model that should execute the requested part of the computational graph (gray box).

- We provide insights and discuss future directions of SL, that are supported by SplitBud.

The framework can foster research in the field while streamlining SL training pipelines into production environments. SplitBud is available at https://github.com/sands-lab/slower.

2 Background

We review the mainstream structure of SL algorithms. As shown in Figure 1, similar to FL, SL algorithms create a model through multiple *training iterations*, which are repeated until a predefined termination condition is satisfied. At a high level, each training iteration consists of five main steps:

1. Client sampling and configuration: At the start of each training iteration, the server selects a subset of active clients and prepares the training parameters for these participants. Training parameters may include the client-side model weights and additional configurations (e.g., learning rate, batch size). The participants receive these instructions and begin the local training (Step 3).

2. Configuration of server model(s): In most SL algorithms, the computational graph is partitioned so that some model layers reside on the server. Thus, at the start of each training iteration, the server also prepares training parameters for these server-side model segments, analogous to the client-side configuration in Step 1. This includes selecting the number of server models, initializing their weights, configuring optimizers, and setting other relevant parameters. The server then instantiates these models on its infrastructure and makes them available to participants.

3. Client training: Each participant trains its portion of the model as instructed in Step 1, while offloading computations to the server. This offloading can involve any part of the computational graph. For example, a client may request the server to perform a forward pass from intermediate embeddings, compute the loss, and backpropagate gradients. Conversely, during inference, the client might request only a forward pass to obtain the target logits. Note that in the former case, the server updates its model weights, while in the latter, it does not.

Since computational graph partitioning varies, we generically refer to these interactions as *client requests*. Upon receiving a request, the server forwards it to the appropriate server model, which executes the requested part of the computational graph. When multiple clients issue requests to the same model, they may be processed in parallel or, as shown in Figure 1, sequentially. Once a client completes training, it informs the server by sending training outcomes, such as updated client-side model weights.

4. Aggregation of client updates: After all participants finish the current round of training, the server aggregates the received updates (e.g., updated client-side model weights). The outcomes of aggregation are then used for the next iteration. This step, employed also in FL, may take various forms, such as simple averaging, weighted averaging, or more sophisticated methods [20, 24, 34].

5. Aggregation of updated server models: If multiple server-side models are instantiated during Step 2, their updates are aggregated at this stage, similar to the process for aggregating client-side updates in Step 4.

The SL protocol presented here is referred to in the literature as *federated* or *parallel* SL to distinguish it from earlier SL formulations, where only one client trains at a time [15]. We, nevertheless, use the term *split learning*, and observe that the one-client-at-a-time training setup is simply a special case of our general formulation, achieved by sampling exactly one client during Step 1.

Furthermore, note that we focus on a cloud-device protocol, considering only SL algorithms that offload computation to a centralized server. In other words, we only examine cases where the computational graph is partitioned between clients and a central server. However, in general, the graph can be divided into multiple segments – for instance, the client may process the initial layers, an edge computing server may handle intermediate layers, and a centralized server may train the uppermost layers [9]. While this setup distributes computation across more resources, it does not alter the computations themselves. In a sense, such a "multi-hop" SL training, where edge servers perform in-network processing, is the SL equivalent of hierarchical FL [25].

3 SplitBud

3.1 Design goals

We design SplitBud to maximize flexibility, ensuring broad applicability across different SL use cases. Specifically, the framework aims to satisfy the following objectives:

- *Arbitrary partitioning of the computational graph*: Any part of the computational graph can be offloaded to the server, whether during training or inference.
- *Flexible data exchange*: Clients can exchange arbitrary data, including embeddings, gradients, logits, and labels.
- *Customizable request-response patterns*: Clients have full control over server interactions, with support for both synchronous and asynchronous invocations.
- *Modular client and server logic*: The framework enforces a clean separation of concerns, allowing client and server logic to be updated independently with minimal overhead.

3.2 Programming abstraction

SplitBud abstracts an SL algorithm into three main components that naturally follow the protocol presented in § 2.

3.2.1 *Client.*

A client executes the logic on an edge device, accessing its private data to train a model. When selected for training, it receives parameters from the server and begins training. When necessary, it offloads part of the computational graph to the server by sending: *a)* the data required for computation, and *b)* an indication of which operations the server should perform.

For example, if the client holds the lower layers of the model being trained and the server holds the upper layers, the client first performs a forward pass through its portion of the model to compute intermediate embeddings. It then sends these embeddings along with the target labels to the server, instructing it to complete the forward pass, compute the loss, and backpropagate until obtaining the gradients with respect to the embeddings. Hence, the server executes the same operations (i.e., the same computational graph) that the client would run if it had sufficient computing power.

3.2.2 *Server Models.*

Server models are the components on the server that execute the computational graph segments requested by clients. Upon receiving a request, they perform the specified computation using the provided data and, if necessary, update their internal state (e.g., model weights). Once the computation is complete, they may return results to the client. Importantly, in SplitBud, server models are *containers* for the models, i.e., server models hold the model being trained as an internal attribute.

3.2.3 *Strategy.*

This component orchestrates training, handling all steps not performed by other components:

- *Step 1*: Samples a subset of clients and prepares their training parameters.
- *Step 2*: Configures training parameters for server models.
- *Step 3*: Routes client requests to the appropriate server model. Routing involves *a)* deciding *which* server model should process a request, as multiple clients and server models may be involved, and *b)* deciding *when* to process a request. For instance, some requests are processed immediately, while others are batched with requests from other clients.
- *Step 4*: Aggregates client updates.
- *Step 5*: Aggregates server updates.

3.3 Programming API

We implement SplitBud as a Python framework, where each component described in § 3.2 is mapped to an abstract class that users override to define the desired algorithm behavior.

For client logic, users override the `Client` class. Specifically, they implement the initialization logic and the `fit` method, which SplitBud invokes in participants at the beginning of each iteration with the training parameters defined by the strategy. Within `fit`, a client trains a model, potentially offloading part of the computational graph to the server. Each client has acces to a `server_model_proxy` attribute, which allows the client to offload arbitrary computations. As mentioned, offloading requires sending the necessary data and specifying the computation to be performed. In code, data is passed as method arguments, while the method name indicates the computation. For example, within `fit`, a client can issue the following statement:

```
logits = server_model_proxy.forward(embeddings=embs)
```

This creates a request containing the required data (`embs`) and the computation to perform (`forward`).

For server models, users override the `ServerModel` abstract class. Specifically, they implement the initialization logic, where they may instantiate the actual model, and the `configure_fit` method, which is called by the framework at the beginning of the training iteration with the training parameters from the strategy. Additionally, the class can have *any* number of methods, which become available to clients. Each method executes a specific computational graph segment. For example, in the previous case, the server model must have a `forward` method accepting a single argument, `embeddings`. Notably, the server model is unaware of which client made a request.

Finally, users define a `Strategy` class to configure clients and server models and aggregate updates from both. The strategy also tracks client requests and determines which server model should handle each request and when.

Example: We illustrate how to implement SplitFed [39] in our framework. A SplitFed client trains the lower layers of the model. It first performs a forward pass on its model segment, then sends the resulting embeddings and labels to the server, and waits for the gradients in return. Once the gradients are received, the client completes the backpropagation step. This logic can be implemented in the client's `fit` method as follows:[2]

```
output = model(batch["images"])
gradient = server_model_proxy.serve_grad_request(
    embeddings=output, labels=batch["labels"]
) # returned value is stored to the `gradient` variable
optimizer.zero_grad()
output.backward(gradient)
optimizer.step()
```

The SplitFed server receives the embeddings and labels sent by clients, performs a forward pass, computes the loss, and backpropagates the loss until obtaining the gradients to be returned to the client. This can be implemented as follows:

```
def serve_grad_request(self, embeddings, labels):
    embeddings.requires_grad_(True)
    output = self.model(embeddings)
    loss = F.cross_entropy(output, labels)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
    return embeddings.grad
```

The developer can then customize the behavior of the strategy by deciding how many clients should train concurrently, aggregating clients' and server models' updates, setting the number of models to be trained on the server, and deciding whether the client's requests should be processed immediately or should be batched with requests from other clients. Thus, by modifying the strategy, one can switch between SplitFed v1 and SplitFed v2 without code changes.

3.4 Implementation

We implement SplitBud as an extension of the Flower FL library [3], adding the necessary logic to support SL-based training. Below, we highlight some key features we introduce.

First, we introduce a `server_model_proxy` attribute for every client. This component establishes and manages a gRPC communication channel with the server, serving as the interface for client-to-server communication. Clients can issue requests using either unary gRPCs, which follow a request-response pattern and are recommended by gRPC's documentation [14], or streaming gRPCs. While SplitBud supports both, we find that streaming gRPCs reduce communication overhead.

[2]The presented code uses PyTorch, but SplitBud is DL framework-agnostic.

Table 1: SL algorithms and lines of code (LoC).

	LoC Client	LoC Server Model
SplitFed v1 [39]	7	8
SplitFed v2 [39]	7	8
U-Shaped SL [40]	13	15
StreamSL [32]	7	7
FedSL [16]	11	7
LocFedMix-SL [28]	17	25
SplitAvg [30]	17	21

Clients can invoke any method on `server_model_proxy` as long as the corresponding method is implemented in the server model, passing any number of arguments. Two special arguments configure the request behavior: `_streams_` specifies whether unary or streaming gRPCs should be used, while `_type_` determines the response handling strategy – clients can either block execution until a response is received or continue processing, handling the asynchronous invocation's return value later.

On the server side, SplitBud employs an `asyncio`-based service to receive and process client requests. The service first deserializes and parses the request. It then provides request metadata to the strategy, which determines the appropriate server model for processing. The service waits until the strategy signals that the request is ready, then triggers computation on the designated server model and returns the results to the client.

4 Preliminary framework evaluation

We now evaluate the framework. Due to page limits, implementation details are in § 3.4. Our goal is not only to show the efficiency of the implementation and the ease with which new algorithms may be implemented, but also to demonstrate the potential benefits of split learning over full on-device training. The code reproducing our results is at https://github.com/BorisRado/split_learning_algorithms.

To demonstrate SplitBud's flexibility, we implement a selection of state-of-the-art SL algorithms. Table 1 lists the algorithms and the number of lines of code required to implement their training logic in SplitBud.[3] Notably, implementing all algorithms requires little effort and minimal code.

4.1 Scalability w.r.t. number of participants

We first analyze how training times vary in a traditional SL setting while varying the number of clients. In SL, all clients share the same server resources. Thus, increasing the number of clients can lead to resource contention, potentially increasing per-round training time due to queuing delays and limited server-side computing capacity. SplitBud enables us to scale the number of clients by simply starting more client instances, with no changes needed to the code.

[3]We do not consider the code for model definition, data loading, etc.

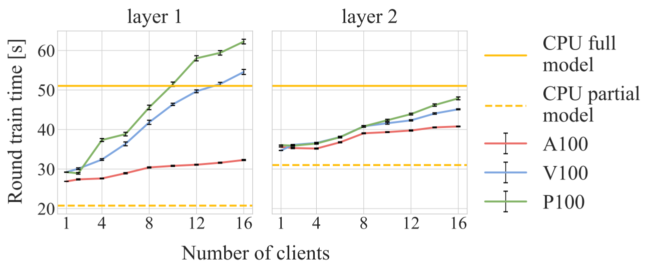

Figure 2: Round train time w.r.t. number of participating clients, size of the client model, and GPU type on the server.

Setting: We train a ResNet-18 model using the SplitFed v1 algorithm, where each client trains a client-side encoder model and a separate server-side classification head. Since clients have the same computing resources (four cores of an Intel Xeon Silver 4112 processor and 8 GB of RAM), they train the same model architecture, specifically the initial ResNet layers up to and including either the first (`layer 1`) or second (`layer 2`) residual block. Thus, the server, equipped with eight CPU cores, 16 GB of RAM, and a GPU accelerator (we vary 3 types), trains the remaining three or two residual blocks, along with the final fully connected layer.

We vary the number of participants from 1 to 16 and report the average training time on a dataset with 400 data points. As baselines, we consider the time required to train the full model locally as in FL ("CPU full model"), and the time to train only the client-side model by performing a forward pass on the client-side model and backpropagating a randomly generated gradient ("CPU partial model").

Results: Figure 2 shows the average time for a training round. We see that SL significantly speeds up training. Training a model on devices takes 51.1 s, while by employing SL with an A100 GPU on the server, the time decreases to 26.8 and 35.4 s depending on whether the client offloads the last three or two residual blocks, respectively. When only one client is training, the process is bottlenecked by the client, as CPU partial model takes 20.7 (`layer1`) and 31.1 s (`layer2`). The increase over this lower bound is caused by increased CPU-GPU data movement, data serialization-deserialization, communication delay, and server training time.

As the number of clients grows, training time increases, influenced by the number of server-trained layers and GPU type. When training a larger portion of the network (`layer1`), V100 and P100 GPUs experience significant slowdowns due to resource saturation. Consequently, SL becomes slower than full-device training when more than 10˘12 clients participate, depending on the GPU. In contrast, the A100 GPU

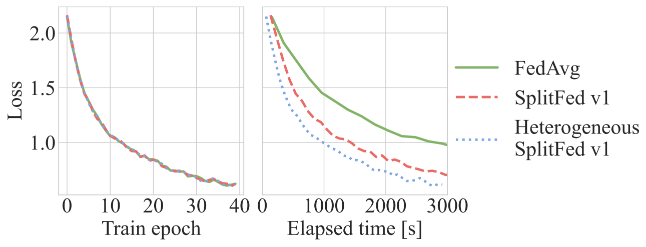

Figure 3: Evaluation of the training loss through training both w.r.t. training epoch and training time.

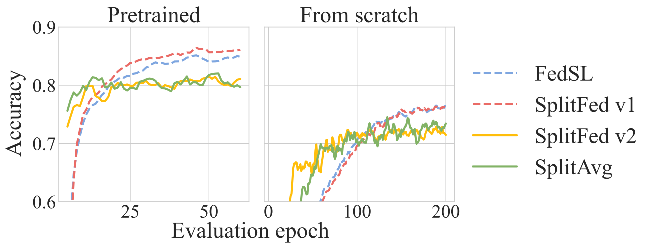

Figure 4: Evolution of the average test accuracy when training from scratch or fine-tuning a pre-trained ResNet 18 model.

handles the load better, with 16 clients requiring only 32.4 s – still significantly faster than full-model training on devices.

Training a smaller segment on the server (`layer2`) reduces GPU load, keeping time increases modest. Even with 16 clients, training is faster than full-model training on devices.

4.2 Training different model sizes on clients

In the previous experiment, we assumed that clients had uniform computing resources. However, in practice, it is often the case that compute resources vary, with some devices being equipped with high computational resources and memory while others have more modest resources. In such cases, different clients can decide to offload different parts of the computational graph to the server. SplitBud allows practitioners to easily implement these use cases.

Setting: We fine-tune a pre-trained ResNet 18 model [17] with three algorithms, namely a) *FedAvg*, in which clients fully train the model locally, b) *SplitFed v1*, in which all clients train until the maxpool layer, and c) *Heterogeneous SplitFed v1*, in which powerful devices train the model locally while weak clients offload the residual blocks and the last fully connected layer to the server. Note, that in all cases, an independent full model is trained for every client. Thus, the next version of the global model is in all cases an average of the models trained in the previous iteration, regardless of whether they have been trained on the client or the server. We run the experiments on CoLExT [5], an experimentation testbed for DCML with 20 clients (8 OrangePi5B, 4 LattePandaDelta3, 2 JetsonAGXOrin, 4 JetsonOrinNano, 2 JetsonXavierNX). In the case of Heterogeneous SplitFed v1 all Jetson devices train the model locally in full while the LattePandas and OrangePis offload part of the computational graph to the server.

Results: Figure 3 reports the evolution of the loss on the clients' datasets over the training round (left plot) and time (right plot). As expected, the loss curves are the same in all variants: since SL *offloads part of the computational graph to the server, it does not alter the computations*. The minor differences arise due to the fact that the random number generators are not synchronized between the clients and the server; thus, the stochastic behavior of the dropout layer is not consistent. This, however, does not yield any significant behavioral differences in terms of convergence.

When training the model fully on-device, the per-round train time[4] is 188.1 ± 0.9 s. This relatively high time is due to stragglers, i.e., devices that take more time to train than others. As FedAvg is a synchronous scheme, it waits until all participants finish training before proceeding to the next iteration. When all clients offload part of the computational graph to the server, the time decreases to 79.6 ± 0.5 s. Despite the significant speed-up, this solution is still sub-optimal, as powerful devices, which could train a model locally sufficiently fast so as to not slow down the training process, still offload computation to the server. This adds unnecessary overhead to the server. Therefore, the optimal solution from a training-time perspective is offloading the computational graph to the server only when necessary as demonstrated by Heterogeneous SplitFed v1, where the per-round time decreases to 55.3 ± 0.4 s.

4.3 Comparing algorithm convergence

We now compare the algorithms' convergence behavior. Note that our aim is not to conclude on absolute algorithm performance as we do not optimize hyperparameters.

Setting: We run the algorithms using 8 clients, each equipped with a dataset with 800 training and 200 test images, and compare the evolution of the average test accuracy throughout training. We consider both the case of training the model from scratch or fine-tuning a pretrained model.

Results: We present in Figure 4 the evolution of test accuracy, focusing on four algorithms for simplicity. We observe that algorithms that involve training a single model on the server (represented by a solid line) converge quickly, but ultimately achieve lower accuracy compared to those that train a separate model for each client on the server (represented by a dashed line). This difference can be attributed to the imbalanced updates discussed in [21, 28].

[4]By per-round training time, we refer to the time elapsed from when training parameters are sent to participants to when the last participant's update is received, as measured by the server.

5 Split Learning – What next?

Overall, SL has received significantly less attention than FL. For instance, a cursory bibliographical search shows that since 2024 over 20,000 papers have mentioned FL, whereas only 1,500 have referenced SL.[5] This disparity is surprising, given that the SL protocol discussed in this paper can be seen as a generalization of FL. In essence, SL allows clients to offload computation to a server, whereas FL restricts them to local training. This is evident in the SL protocol formulated in § 2 – if clients do not offload to the server in step 3, the protocol reduces to FL, making steps 2 and 5 unnecessary.

We argued that SL can be seen as redistributing computation across entities *without altering the computations themselves.* In this context – assuming each client trains its own server-side model – SL can complement *any* FL algorithm by reducing the computational burden on the client side. Importantly, this redistribution does not affect the convergence behavior of the algorithms (cf. Figure 3).

We identify a gap in the literature regarding the number of models trained on the server. As we also demonstrated (cf. Figure 4), having all clients train a shared server-side model can lead to lower accuracy than assigning each client its own model. However, the latter approach is memory-intensive. A promising research direction is optimizing the number of server-side models and, more broadly, exploring strategies to balance resource efficiency with model performance. Ultimately, the goal is to make SL scalable and effective for large deployments involving thousands or millions of clients.

From a privacy standpoint, SL is less protective than FL, as the server in SL has access to strictly more data. In FL, the server sees only the model weights before and after client training. In addition to this, SL exposes also the evolution of server-side weights and intermediate data embeddings. Moreover, in algorithms that do not use a U-shaped architecture, clients must also transmit target labels to the server. Sharing labels is particularly problematic in language modeling, where sharing the target labels implies sharing all the input data due to the nature of the next-token prediction training task. However, to our knowledge, the U-shaped architecture has only been applied with a single client at a time [31, 40] or when each client trains its own server model [33]. Both approaches face severe limitations, as we discussed. However, there is no technical reason preventing one from developing a more general SL algorithm with U-shaped architecture. We leave the exploration of this to future research.

The benefits of SL extend towards democratization of machine learning. On the one hand, SL enables resource-constrained clients to train large models. On the other, it facilitates a more balanced distribution of compute resources. Unlike FL, where clients perform all the computation while the server remains underutilized, and centralized training, where the server performs all the work while edge devices are idle, SL allows both clients and the server to contribute data and compute proportionally to their resources.

We hope that SplitBud will foster research in these directions, making FL and SL coexist and strengthen each other.

[5]Source: Google Scholar. Values checked on February 5, 2025.

6 Related work

Parallel Split Learning combines FL and SL, allowing multiple clients to concurrently train while the server assists in training and aggregation [12, 39]. Variants differ in how server-side models are managed: SplitFed v1 [39] maintains separate heads per client, SplitFed v2 [39] shares a common head, and SFGL [12] trains multiple server-side models. SplitFed v3 [13] addresses non-IID data by keeping client-side model segments private. Recent works have tackled challenges such as communication overhead [2, 6, 16], server-client update imbalance [21, 28], heterogeneous model partitioning [9, 35], and security aspects of training [1].

Distributed Machine Learning Frameworks: Several frameworks support multi-node training. Petals [4] enables split training for large language models but restricts clients to train only the layers they own. FusionLLM [37] supports arbitrary models, automatic differentiation, and heterogeneous software. Both frameworks focus on implementing *pipeline parallelism* [27], using data from a single client. In contrast, SplitBud is designed for SL algorithms, supporting data aggregation across clients, concurrent multi-client training, and simultaneous model training on the server.

7 Conclusion

We introduced SplitBud, a flexible framework for implementing and evaluating SL algorithms. By unifying SL implementations, SplitBud facilitates systematic exploration of SL's strengths, limitations, and integration with FL. Our experiments demonstrated its versatility across multiple SL algorithms, while our analysis highlighted key challenges faced by SL, including privacy, server-side model management, and resource efficiency.

Acknowledgments

This publication is based upon work supported by the King Abdullah University of Science and Technology Research Funding (KRF) under Award No. ORA-CRG2021-4699. The research presented in this paper was partly funded by Slovenian Research Agency projects "Context-Aware On-Device Approximate Computing" (J2-3047) and "approXimation for adaptable distributed artificial intelligence" (N2-0393).

References

[1] Sharif Abuadbba, Kyuyeon Kim, Minki Kim, Chandra Thapa, Seyit Ahmet Çamtepe, Yansong Gao, Hyoungshick Kim, and Surya Nepal. 2020. Can We Use Split Learning on 1D CNN Models for Privacy Preserving Training?. In *ASIA CCS*.

[2] Ahmad Ayad, Melvin Renner, and Anke Schmeink. 2021. Improving the Communication and Computation Efficiency of Split Learning for IoT Applications. In *IEEE Global Communications Conference, GLOBECOM 2021*.

[3] Daniel J. Beutel, Taner Topal, Akhil Mathur, Xinchi Qiu, Titouan Parcollet, and Nicholas D. Lane. 2020. Flower: A Friendly Federated Learning Research Framework. (2020).

[4] Alexander Borzunov, Dmitry Baranchuk, Tim Dettmers, Maksim Riabinin, Younes Belkada, Artem Chumachenko, Pavel Samygin, and Colin Raffel. 2023. Petals: Collaborative Inference and Fine-tuning of Large Models. In *ACL*.

[5] Janez Božič, Amandio R Faustino, Boris Radovič, Marco Canini, and Veljko Pejović. 2024. Where is the Testbed for my Federated Learning Research? (2024).

[6] Ayush Chopra, Surya Kant Sahu, Abhishek Singh, Abhinav Java, Praneeth Vepakomma, Vivek Sharma, and Ramesh Raskar. 2021. AdaSplit: Adaptive Trade-offs for Resource-constrained Distributed Deep Learning. (2021).

[7] Enmao Diao, Jie Ding, and Vahid Tarokh. 2021. HeteroFL: Computation and Communication Efficient Federated Learning for Heterogeneous Clients. In *ICLR*.

[8] Tao Fan, Yan Kang, Guoqiang Ma, Weijing Chen, Wenbin Wei, Lixin Fan, and Qiang Yang. 2023. FATE-LLM: A Industrial Grade Federated Learning Framework for Large Language Models. (2023).

[9] Wenhao Fan, Penghui Chen, Xiongfei Chun, and Yuan'an Liu. 2025. MADRL-based model partitioning, aggregation control, and resource allocation for cloud-edge-device collaborative split federated learning. *IEEE Trans. Mob. Comput.* (2025).

[10] Flower. 2024. Introducing FlowerLLM. https://flower.ai/blog/2024-03-14-introducing-flowerllm/. Accessed: 2 Feb 2025.

[11] Yansong Gao, Minki Kim, Sharif Abuadbba, Yeonjae Kim, Chandra Thapa, Kyuyeon Kim, Seyit Ahmet Çamtepe, Hyoungshick Kim, and Surya Nepal. 2020. End-to-End Evaluation of Federated Learning and Split Learning for Internet of Things. In *International Symposium on Reliable Distributed Systems, SRDS 2020*.

[12] Yansong Gao, Minki Kim, Chandra Thapa, Alsharif Abuadbba, Zhi Zhang, Seyit Camtepe, Hyoungshick Kim, and Surya Nepal. 2022. Evaluation and Optimization of Distributed Machine Learning Techniques for Internet of Things. *IEEE Trans. Computers* (2022).

[13] Manish Gawali, C. S. Arvind, Shriya Suryavanshi, Harshit Madaan, Ashrika Gaikwad, K. N. Bhanu Prakash, Viraj Kulkarni, and Aniruddha Pant. 2021. Comparison of Privacy-Preserving Distributed Deep Learning Methods in Healthcare. In *MIUA*.

[14] gRPC documentation. 2025. Performance Best Practices. https://grpc.io/docs/guides/performance. Accessed: 27 Jan 2025.

[15] Otkrist Gupta and Ramesh Raskar. 2018. Distributed learning of deep neural network over multiple agents. *J. Netw. Comput. Appl.* (2018).

[16] Dong-Jun Han, Hasnain Irshad Bhatti, Jungmoon Lee, and Jaekyun Moon. [n. d.]. Accelerating federated learning with split learning on locally generated losses. https://fl-icml.github.io/2021/papers/FL-ICML21_paper_6.pdf. Accessed: 2023-11-16.

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*.

[18] Samuel Horváth, Stefanos Laskaridis, Mário Almeida, Ilias Leontiadis, Stylianos I. Venieris, and Nicholas D. Lane. 2021. FjORD: Fair and Accurate Federated Learning under heterogeneous targets with Ordered Dropout. In *NeurIPS*.

[19] Sohei Itahara, Takayuki Nishio, Yusuke Koda, Masahiro Morikura, and Koji Yamamoto. 2023. Distillation-Based Semi-Supervised Federated Learning for Communication-Efficient Collaborative Training With Non-IID Private Data. *IEEE Trans. Mob. Comput.* (2023).

[20] Kairouz, Peter and McMahan, H Brendan and Avent, Brendan and Bellet, Aurélien and Bennis, Mehdi and Bhagoji, Arjun Nitin and Bonawitz, Kallista and Charles, Zachary and Cormode, Graham and Cummings, Rachel and others. 2021. Advances and Open Problems in Federated Learning. *Foundations and Trends® in Machine Learning* (2021).

[21] Mohammad Kohankhaki, Ahmad Ayad, Mahdi Barhoush, and Anke Schmeink. 2024. Parallel Split Learning with Global Sampling. (2024).

[22] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated Learning: Strategies for Improving Communication Efficiency. (2016).

[23] Daliang Li and Junpu Wang. 2019. FedMD: Heterogenous Federated Learning via Model Distillation. (2019).

[24] Tao Lin, Lingjing Kong, Sebastian U. Stich, and Martin Jaggi. 2020. Ensemble Distillation for Robust Model Fusion in Federated Learning. In *NeurIPS*.

[25] Lumin Liu, Jun Zhang, Shenghui Song, and Khaled B. Letaief. 2020. Client-Edge-Cloud Hierarchical Federated Learning. In *ICC*.

[26] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS*.

[27] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *SOSP*.

[28] Seungeun Oh, Jihong Park, Praneeth Vepakomma, Sihun Baek, Ramesh Raskar, Mehdi Bennis, and Seong-Lyun Kim. 2022. LocFedMix-SL: Localize, Federate, and Mix for Improved Scalability, Convergence, and Latency in Split Learning. In *WWW*.

[29] Haseena Rahmath P, Vishal Srivastava, Kuldeep Chaurasia, Roberto Gonçalves Pacheco, and Rodrigo S. Couto. 2025. Early-Exit Deep Neural Network - A Comprehensive Survey. *ACM Comput. Surv.* (2025).

[30] Shraman Pal, Mansi Uniyal, Jihong Park, Praneeth Vepakomma, Ramesh Raskar, Mehdi Bennis, Moongu Jeon, and Jinho Choi. 2021. Server-Side Local Gradient Averaging and Learning Rate Acceleration for Scalable Split Learning. (2021).

[31] Maarten G. Poirot, Praneeth Vepakomma, Ken Chang, Jayashree Kalpathy-Cramer, Rajiv Gupta, and Ramesh Raskar. 2019. Split Learning for collaborative deep learning in healthcare. (2019).

[32] Boris Radovič, Mohammed Aljahdali, Marco Canini, Veljko Pejović, and Zuhair Khayyat. 2024. Train your cake and eat it too! Repurposing collaborative training to tailor LLMs to private data without sharing. (2024).

[33] Boris Radovič, Marco Canini, and Veljko Pejović. 2024. Review and comparative evaluation of resource-adaptive collaborative training for heterogeneous edge devices. *ACM Trans. Model. Perform. Eval. Comput. Syst.* (2024).

[34] Sashank J. Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečný, Sanjiv Kumar, and Hugh Brendan McMahan. 2021. Adaptive Federated Optimization. In *ICLR*.

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

[36] Yue Tan, Guodong Long, Lu Liu, Tianyi Zhou, Qinghua Lu, Jing Jiang, and Chengqi Zhang. 2022. FedProto: Federated Prototype Learning across Heterogeneous Clients. In *AAAI*.

[37] Zhenheng Tang, Xueze Kang, Yiming Yin, Xinglin Pan, Yuxin Wang, Xin He, Qiang Wang, Rongfei Zeng, Kaiyong Zhao, Shaohuai Shi, Amelie Chi Zhou, Bo Li, Bingsheng He, and Xiaowen Chu. 2024. FusionLLM: A Decentralized LLM Training System on Geo-distributed GPUs with Adaptive Compression. (2024).

[38] Surat Teerapittayanon, Bradley McDanel, and H. T. Kung. 2016. BranchyNet: Fast inference via early exiting from deep neural networks. In *ICPR*.

[39] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, Seyit Camtepe, and Lichao Sun. 2022. SplitFed: When Federated Learning Meets Split Learning. In *AAAI*.

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

[41] Pablo Villalobos, Anson Ho, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, and Marius Hobbhahn. 2024. Will we run out of data? Limits of LLM scaling based on human-generated data. (2024).

[42] Pablo Villalobos, Jaime Sevilla, Tamay Besiroglu, Lennart Heim, Anson Ho, and Marius Hobbhahn. 2022. Machine Learning Model Sizes and the Parameter Gap. (2022).

[43] Chuhan Wu, Fangzhao Wu, Lingjuan Lyu, Yongfeng Huang, and Xing Xie. 2022. Communication-efficient federated learning via knowledge distillation. *Nat. Commun.* (2022).