Train your cake and eat it too! Repurposing collaborative training to tailor LLMs to private data without sharing

Boris Radovič [*1,2] **Mohammed Aljahdali** [*1] **Marco Canini** [1] **Veljko Pejović** [2] **Zuhair Khayyat** [3]

Abstract

In the emerging field of large language models (LLMs), a significant challenge arises when organizations with vast datasets lack the computational resources to independently train and fine-tune models. This issue stems from privacy, compliance, and resource constraints: organizations cannot share their sensitive data but still need external computational assistance for model training. In this paper, we implement, enhance, and empirically compare several methods, including Split Learning (SL) and select Federated Learning (FL) methods, which enable data-rich yet compute-poor clients to offload LLM training without sharing raw data. Our study evaluates these methods across multiple dimensions, including model quality and training time.

1. Introduction

Large Language Models (LLMs) are increasingly gaining traction, finding applications in myriad of domains, including programming (Chen et al., 2021), biomedicine (Gu et al., 2021), and question answering (Yang et al., 2022). With model sizes often surpassing several billion parameters, successful LLMs are typically trained in data centers equipped with specialized hardware, entailing significant investments in terms of finances, energy consumption, and time (Luccioni et al., 2023). Considering their size, these models require a large corpus of data for training; so, most LLMs leverage swaths of publicly available data collected from heterogeneous data sources, such as web-scraped content, which also makes the so-obtained models versatile problem solvers across several tasks.

Tailoring an LLM with the help of domain-specific private data – the focus of our work – promises to improve the model's performance in particular use cases. Many companies, for instance, rely on models trained on their customer data as a cornerstone of their business, necessitating a model specifically designed and trained on their proprietary data. In many cases, training such models on in-house infrastructure is infeasible due to very high computational costs. To cope, it is indeed common to start from publicly available pre-trained LLMs and fine-tune all model parameters on the downstream task.[1] However, very large models render this strategy impractical for small organizations due to high hardware requirements. While parameter-efficient fine-tuning lowers the hardware requirements (Houlsby et al., 2019; Hu et al., 2022; He et al., 2022), we consider it an undesirable approach towards our goal of tailoring LLM on private data, beyond the specific reach of fine-tuning on a specialized dataset or downstream task (Wang et al., 2024).

Thus, organizations face a dilemma. They can upload their private data to the cloud and centrally[2] train the model employing the cloud's vast computational resources. However, data sharing with cloud platforms may present challenges regarding data privacy, sovereignty and legal compliance (Truong et al., 2021). Alternatively, organizations may upgrade their private infrastructure, but this option may be cost-prohibitive. Under such circumstances, there is a dire need for methods capable of harnessing the combined computational capabilities of both the local infrastructure and cloud providers *without* sharing raw data.

To meet data privacy requirements in other contexts, collaborative training paradigms such as Federated Learning (FL) (Bonawitz et al., 2019; McMahan et al., 2017; Konečný et al., 2016) and Split Learning (SL) (Poirot et al., 2019; Vepakomma et al., 2018; Gupta & Raskar, 2018) have emerged. In the former, a "federation" of clients collaborates to train a single model by sharing locally computed model updates. In the latter case, SL partitions the model horizontally into two or more segments which are trained in an end-to-end fashion.

[*]Equal contribution [1]KAUST [2]University of Ljubljana [3]Lucidya. Correspondence to: Boris Radovič <boris.radovic@kaust.edu.sa>, Mohammed Aljahdali <mohammed.kaljahdali@kaust.edu.sa>.

[1]We broadly refer to *LLM training* as the process of updating model parameters using an organization's private data, whether this involves fine-tuning a pre-trained LLM or (pre-)training an LLM from scratch.

[2]Standard (or centralized) training refers to the process of training a model on a cluster with sufficient computational resources, where the cluster nodes have direct access to the raw training data.

Existing FL methods are not directly applicable to the discussed *one-client* setup where a single client (i.e., an organization) is the data owner and wants to use a public cloud's computational capabilities to train an LLM. This means model-sharing FL methods in (McMahan et al., 2017; Li et al., 2020) are only designed to improve model convergence in cases where *multiple* clients wish to collaborate. Furthermore, in most FL methods, the high computational capacity of the server remains untapped, as the server is relegated to performing coordination tasks. Some FL methods such as FedGKT (He et al., 2020) and FSL (Han et al., 2021) avoid this limitation by training smaller models on the clients and a larger model on the server.

SL methods allow a single client to leverage the computational capabilities of the server to train a model, that is larger than the one they could train on their own. While SL does not alter the training procedures and thus ensures the same model quality as standard training, this approach incurs high network utilization and potentially low resource utilization – we discuss these issues throughout the paper.

To the best of our knowledge, these methods have never been applied to the one-client setup. Specifically, neither FL methods nor SL have been explored for offloading the training of LLMs. Given these considerations, our study aims to explore the following key questions:

- Can FedGKT and FSL train a model that matches the *quality* of a model trained centrally or using SL?
- Among these methods, what is the most *time efficient* method for training LLMs in the one-client setup?

To address these questions, our main contributions are:

- We develop an *extensible framework* that enables practitioners to experiment with and deploy all considered methods in real-world settings.
- We discuss *optimizations that enhance the efficiency* of the methods in the one-client scenario and generally.
- We *evaluate* FedGKT, FSL, and three SL variants, comparing their *model quality and throughput*.

2. Repurposing collaborative training

2.1. Background

We consider a setup in which there is a single client that has a private dataset $D = \{(x_j, y_j)\}_{j=1}^{|D|}$ where x_j represents the j-th input sample, y_j the corresponding ground-truth label, and $|D|$ the cardinality of the private dataset. The final goal is to obtain a model that minimizes the following objective: $\min_{w \in \mathbb{R}^d} \frac{1}{|D|} \sum_{j=1}^{|D|} l(f_s(w_s, f_c(w_c, x_j)), y_j)$, where $f_i(w_i, x)$ with i either 'c' or 's' for client and server, respectively is the output of the model parametrized with weights $w = w_c \cup w_s$ when presented input x and task-specific loss l. The main assumption in this work is that the client cannot train a large enough model w on its own, thus it needs to *offload* a portion of the training workload to a server.

In all of the methods we discuss the client sends intermediate features $f_c(w_c, x_j)$ and the target label y_j to the server,[3] where w_c is the client weights or a subset of such weights. Moreover, in all of the methods there is a model split problem, where the user has to decide how many layers the client model and how many layers the server model should have.

2.2. Applying existing methods

Split Learning. Similar to pipeline parallelism (Huang et al., 2019), SL (Gupta & Raskar, 2018; Vepakomma et al., 2018; Poirot et al., 2019) horizontally partitions a model $w = w_1, \ldots, w_k$ into k subsequent components that are executed sequentially. In other words, the model output is iteratively defined as $e_i = f_i(w_i; e_{i-1})$, with e_0 representing the input features. In the most basic scenario with $k = 2$, the first L layers of the model are located on the client, while the remaining layers are on the server. In such a setup the server computes the loss, hence the client is required to share the target labels.

In the U-shaped architecture (Vepakomma et al., 2018), the model is partitioned into $k = 3$ segments, with the client holding both w_1 and w_3. This arrangement shifts the computation of the loss to the client. While eliminating the need for the client to share target labels, this architectural variation comes with additional communication overhead, as for any gradient update, four passes over the network are required – two during the forward and two during the backward pass.

We only consider $k = 2$ (Plain SL) and $k = 3$ (U-shaped SL) since even if the model were partitioned into multiple segments on the server, these are perceived as a single logical entity by the client.

We note that PETALS (Borzunov et al., 2023), a platform for collaborative fine-tuning and inference for large models, during the fine-tuning use case, can be viewed as an instance of U-shaped SL. A PETALS client maps input data to intermediate features, which are sent to a chain of servers. These servers process the features through a sequence of layers and return the final intermediate features to the client, which completes the model's forward pass, computes the loss and starts the backward pass. The model parameters at the servers must remain frozen and only client-side parameters are trainable. Instead, we allow for all parameters to be updated.

[3] An exception is the U-shaped SL, which we discuss later.

FedGKT. In FedGKT (He et al., 2020) a client model w_c and a server model w_s are trained in parallel. The client model is logically separated into an encoder $w_{c,e}$ and a classification head $w_{c,h}$. As for the server model, it has many layers followed by a classification head. The server model expects intermediate features $e_i = f(w_{c,e}, x_i)$ from the client as an input. Also, the server receives the ground-truth label y_j and client's logits $f(w_{c,h}, f(w_{c,e}, x_i))$. Both the client and the server update their models by locally computing a loss that consists of the task-specific term, e.g., the cross-entropy loss, and a knowledge distillation term, in which the logits of the other model serve as knowledge teacher. Note that the client starts using the distillation term from the second round onwards, as it does not have access to the server logits in the first round. At inference time, the client can use either its model w_c or its encoder $w_{c,e}$ in conjunction with the server model w_s.

FSL. FSL (Han et al., 2021), like FedGKT, concurrently trains both a client model and a server model, both equipped with a classification head. However, unlike FedGKT, in FSL, there is no knowledge distillation term in either the client or the server loss. As a result, in FSL, the client model is updated independently of the server model.

Application to LLM. In all the methods discussed, the client retains control over the lower layers of the model undergoing training. In the context of LLMs, these layers include the embedding table and one or more transformer blocks. In some algorithms (such as FedGKT and FSL), the client also possesses an auxiliary classification head. Conversely, the server model always consists of one or more transformer blocks and, with the exception of the U-shaped SL method, also a final output layer (classification head).

3. Optimizations

In this section, we discuss some implementation optimizations that can enhance the system efficiency without compromising the model's final performance.

Compressing token embeddings. Network communication plays a crucial role in all of the above-discussed methods. A straightforward implementation would necessitate the client sending $n \cdot h$ floating-point values for each training example to the server, where n is the sequence length operated on by the LLM and h is the embedding dimension. One potential strategy to reduce network traffic when training LLMs is letting the client share only the embeddings of non-padding tokens. Specifically, as LLMs process fixed-length sequences, sequences shorter than the target length are padded to match it. Since padded tokens are not attended to during the forward computation, clients can avoid transmitting them and the server can set such embeddings to $\vec{0}$. A similar optimization can be applied when sending gradients from the server to the client. We explore the benefits of this lossless compression in Section 4.3.

Overlapping computation. In SL, the client and server training loops are tightly coupled. That is, after sending data to the server, the client waits for the server's response, and vice versa. Similarly, in FedGKT, both nodes experience long idle times due to round-based communication, where the client and server alternately train.

FSL avoids this limitation because the client trains its model solely on the local data. This property is desirable because the client does not need to wait for the server to do the training. Therefore, in FSL the client communicates the intermediate features immediately after the forward pass, before the backward pass. Thus, an overlap between the client and server computations is achieved.

We observe that FedGKT can be adapted to obtain the same property as FSL. We modify the client logic to send features, logits, and target labels to the server as soon as they are computed. This modification allows the server to start training without waiting for the client to finish its training.

In SL, overlapping computation between the client and server is feasible when all client parameters are frozen – this is sometimes desirable when fine-tuning a pre-trained model. In such a case, as the client does not require gradient data from the server, it can continue training without waiting for the server's response. We term this SL variant with overlapping computation as "StreamSL."

4. Evaluation

We evaluate various methods for training a model in the one-client setup. Given the client's constrained resources, the goal is for these methods to attain a *model quality* comparable to standard training within an *efficient training time*. Since the client may rent cloud resources for the method's execution, training time directly affects the overall cost. To this end, we conduct various experiments to assess the model quality (accuracy) and evaluate system performance (training throughput and communication volume).

We find that to date, there exists no framework that facilitates implementing and deploying the considered algorithms. Therefore, to make a fair assessment of system performance, we develop a flexible extension to Flower (Beutel et al., 2022), one of the most popular and widely used FL frameworks. The need for our extension arises because in Flower, clients cannot invoke server functions. Furthermore, Flower, being designed for FL settings, does not include the concept of a server-side model. The extension allows users to effortlessly implement and deploy all the methods described in Section 2.2. We hence utilize this extension throughout Section 4. We provide a detailed discussion of the extension's

Figure 1. Test accuracy comparison between FedGKT and FSL relative to "Standard training X", where X is the number of layers in the model. "Node X" denotes the X-layer model trained on the respective node, e.g., Server 23 refers to the server model with 23 layers.

architecture and usage examples in Appendix A.

4.1. Model quality

To assess **whether FedGKT and FSL can achieve comparable or superior model quality compared to standard training**, we conduct a comprehensive comparison of the accuracies achieved by all of these methods.[4] We use the BERT model architecture for all of our experiments (Devlin et al., 2019). Specifically, we assume the client can locally train a small model with either 1 or 3 layers.[5] The accuracy attained by such models serves as a lower bound, as failure to meet this threshold would make using these methods impractical. As an upper bound, we explore the training of 12- (BERT *base*) or 24-layer (BERT *large*) models, which would exceed the capacity for client-side training, hence it serves an artificial upper bound. Throughout these experiments, we initialize the layers with either BERT *base* or BERT *large* pre-trained model weights. For FSL and FedGKT, the model weights are partitioned across the two nodes. For instance, when working with pre-trained 12-layer model with 3 client layers, the embedding table and the first 3 layers are managed by the client, while the remaining 9 layers are placed on the server. The classification head in all cases is initialized randomly. Our evaluation focuses on sequence classification tasks across three datasets: IMDB reviews (Maas et al., 2011), Ag News, and Yelp reviews (Zhang et al., 2015).

In all experiments, we execute the method for 5 training epochs and report the test accuracy of the model (server model in FedGKT and FSL) checkpoints with the highest validation accuracy. We present the accuracies achieved using the BERT *large* model in this section and include results for the BERT *base* model in Appendix B. As both FSL and FedGKT involve training two models (client & server), we report the accuracy achieved by each model separately in Figure 1. We observe that the accuracy achieved by the client model is consistently lower than that of the server model, which is expected due to the deeper model architecture used on the server. Furthermore, FedGKT generally exhibits slightly higher accuracies compared to FSL, most notably in the client model due to the use of knowledge distillation. Still, the server model accuracy differences are often marginal. However, the key observation is that both **FedGKT and FSL consistently achieve similar model quality to standard training** across all experiments.

4.2. Throughput

We next analyze **which method shows the shortest training time.** We quantify this by measuring throughput, which is defined as the number of batches processed per unit of time and thus is inversely proportional to training time. For the experiment, we use a high-capacity server equipped with an A100 GPU and a low-capacity client with a P6000 GPU, connected via a 1 Gbps link. Figure 2 reports the throughput of the five considered methods while varying the number of layers trained on both the client and the server.

Methods with non-overlapping computation. The Plain SL and U-shaped SL methods do not involve overlapping computations and consequently achieve the lowest throughput. Among the two, the U-shaped variant exhibits slightly lower throughput due to increased network communication and the fact that the final fully connected layer is executed on the client.

Considering the speed disparity between the client and server, maximizing throughput in these methods requires maximizing the number of layers executed on the server. For instance, in Figure 2 we see that throughput is consistently

[4] SL is equivalent to centralized training.

[5] We refer to a transformer block and in this case a BERT block as a layer for simplicity.

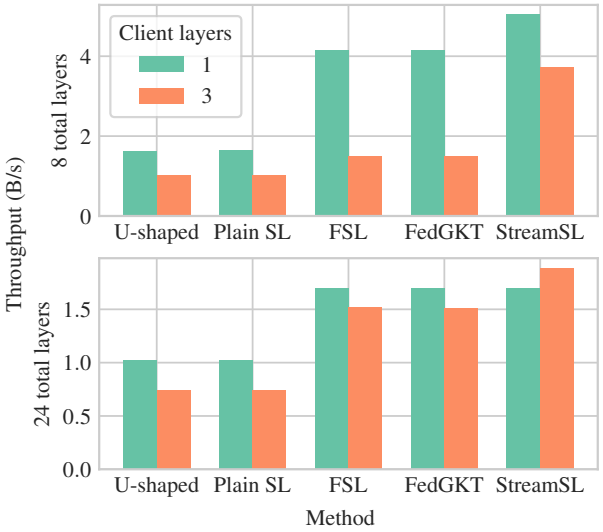

Figure 2. Throughput achieved by the methods using 8 and 24 layers, of which 1 or 3 are held by the client. The batch size B is set to 32.

higher when the client operates one layer compared to three layers. However, adjusting the number of layers executed on the client may impact the risk of privacy leakage (Abuadbba et al., 2020). Finally, note that varying the number of layers does not impact the final accuracy of the trained model.

Methods with overlapping computation. The observed patterns in the FSL, FedGKT, and StreamSL methods can be attributed to the presence of a bottleneck node, which is slower than the other node and consequently prevents the overall system from achieving higher throughput. Specifically, when the server experiences low throughput due to training a high number of layers, the slower processing of the client does not impact the overall training speed. For example, when training a model with 24 layers, the throughput of the three methods is very similar and exhibits minimal variance with respect to the number of client layers, as the server is the bottleneck. In contrast, when training a model with 8 layers, increasing the number of client layers from 1 to 3 leads to a substantial drop in throughput for the FSL and FedGKT methods because the client becomes the bottleneck. This drop is less pronounced for StreamSL, as in this method, the client only performs the forward pass and therefore has a much smaller computational burden.

Methods with overlapping computation achieve the most effective usage of the available resources when there is no single bottleneck node – meaning, both the client and server process data at the same pace, thereby avoiding idle time. Yet, even if such a perfect balance is not achieved, **methods with overlapping computation still achieve the highest training throughput**. However, unlike the two SL variants previously discussed, in these methods varying the number of client layers impacts the quality (accuracy) of the model as observed in Figure 1.

Table 1. Average num. of communicated tokens and Std. Dev.

Dataset	Num. Communicated Tokens
IMDB	280 ± 140
Yelp Reviews	178 ± 136
Ag News	57 ± 21

4.3. Communication volume

We test the lossless compression presented in Section 3 and observe that it **significantly reduces the amount of communicated data without impacting model performance**. In our experiments, we set the context size to 512, hence a naïve approach would require sending 512 768-dimensional token embeddings for each training example, totaling $512 \cdot 768 \approx 40$k floating-point values. With the compression, however, the nodes only exchange as many token embeddings as the length of the input sequence. As shown in Table 1, this reduces the number of exchanged embeddings – and hence floating-point values – by 45% to 88%. While reducing the context size could alleviate the issue, it might lead to information loss and consequently compromise model performance.

Reducing the amount of exchanged data decreases communication time due to decreased data transmission over the network and reduced serialization and deserialization overheads at the nodes. However, while the compression affects the throughput of all methods, its impact varies. Specifically, methods with overlapping computation may not benefit as much as their non-overlapping counterparts, as network latency can be masked by computation time.

5. Conclusion

Our effort represents an initial step toward enabling the training of LLMs in the one-client setup, where a compute-constrained client wishes to leverage a powerful server in a privacy-preserving manner. Specifically, to preserve the client's data privacy while allowing the client to fine-tune an LLM, we considered various SL variants and adapted two FL methods, FedGKT and FSL, to fit the problem formulation. We implemented these methods with several computation and communication optimizations that maintain the method's performance and compared them in terms of accuracy and training time. Notably, our findings reveal that both FedGKT and FSL achieve accuracy on par with centralized training and SL and at the same time significantly reduce training time over SL due to the overlapping computation between the client and server nodes.

Acknowledgments

This work was supported by the SDAIA-KAUST Center of Excellence in Data Science and Artificial Intelligence (SDAIA-KAUST AI) and by the Slovenian Research Agency (research projects J2-3047 and P2-0098). We are thankful to Asaad Mohammedsaleh for his contributions during the early phase of the project.

References

Abuadbba, S., Kim, K., Kim, M., Thapa, C., Camtepe, S. A., Gao, Y., Kim, H., and Nepal, S. Can We Use Split Learning on 1D CNN Models for Privacy Preserving Training? In *ASIA CCS*, 2020. URL https://doi.org/10.1145/3320269.3384740.

Beutel, D. J., Topal, T., Mathur, A., Qiu, X., Fernandez-Marques, J., Gao, Y., Sani, L., Li, K. H., Parcollet, T., de Gusmão, P. P. B., and Lane, N. D. Flower: A Friendly Federated Learning Research Framework, 2022. URL https://arxiv.org/abs/2007.14390.

Bonawitz, K., Eichner, H., Grieskamp, W., Huba, D., Ingerman, A., Ivanov, V., Kiddon, C., Konečný, J., Mazzocchi, S., McMahan, H. B., Overveldt, T. V., Petrou, D., Ramage, D., and Roselander, J. Towards Federated Learning at Scale: System Design. In *MLSys*, 2019. URL https://proceedings.mlsys.org/paper_files/paper/2019/file/7b770da633baf74895be22a8807f1a8f-Paper.pdf.

Borzunov, A., Baranchuk, D., Dettmers, T., Riabinin, M., Belkada, Y., Chumachenko, A., Samygin, P., and Raffel, C. Petals: Collaborative Inference and Fine-tuning of Large Models. In *ACL*, 2023. URL https://aclanthology.org/2023.acl-demo.54.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating Large Language Models Trained on Code, 2021. URL https://arxiv.org/abs/2107.03374.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*, 2019. URL https://aclanthology.org/N19-1423.

Gu, Y., Tinn, R., Cheng, H., Lucas, M., Usuyama, N., Liu, X., Naumann, T., Gao, J., and Poon, H. Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing. *ACM Trans. Comput. Healthcare*, 3(1), 2021. doi: 10.1145/3458754. URL https://doi.org/10.1145/3458754.

Gupta, O. and Raskar, R. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116, 2018. URL https://doi.org/10.1016/j.jnca.2018.05.003.

Han, D.-J., Bhatti, H. I., Lee, J., and Moon, J. Accelerating federated learning with split learning on locally generated losses. In *FL-ICML*, 2021. URL https://fl-icml.github.io/2021/papers/FL-ICML21_paper_6.pdf.

He, C., Annavaram, M., and Avestimehr, S. Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge. In *NeurIPS*, 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/a1d4c20b182ad7137ab3606f0e3fc8a4-Paper.pdf.

He, J., Zhou, C., Ma, X., Berg-Kirkpatrick, T., and Neubig, G. Towards a unified view of parameter-efficient transfer learning. In *ICLR*, 2022.

Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-Efficient Transfer Learning for NLP. In *ICML*, 2019. URL https://proceedings.mlr.press/v97/houlsby19a.html.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. LoRA: Low-Rank Adaptation of Large Language Models. In *ICLR*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *NeurIPS*, 2019. URL https://papers.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

Konečný, J., McMahan, H. B., Yu, F. X., Richtárik, P., Suresh, A. T., and Bacon, D. Federated Learning: Strategies for Improving Communication Efficiency, 2016. URL https://arxiv.org/abs/1610.05492.

Li, T., Sahu, A. K., Zaheer, M., Sanjabi, M., Talwalkar, A., and Smith, V. Federated Optimization in Heterogeneous Networks. In *MLSys*, 2020. URL https://proceedings.mlsys.org/paper_files/paper/2020/file/1f5fe83998a09396ebe6477d9475ba0c-Paper.pdf.

Luccioni, A. S., Viguier, S., and Ligozat, A.-L. Estimating the Carbon Footprint of BLOOM, a 176B Parameter Language Model. *Journal of Machine Learning Research*, 24(253), 2023. URL http://jmlr.org/papers/v24/23-0069.html.

Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., and Potts, C. Learning Word Vectors for Sentiment Analysis. In *ACL*, 2011. URL http://www.aclweb.org/anthology/P11-1015.

McMahan, B., Moore, E., Ramage, D., Hampson, S., and Arcas, B. A. y. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *AISTATS*, 2017. URL https://proceedings.mlr.press/v54/mcmahan17a.html.

Poirot, M. G., Vepakomma, P., Chang, K., Kalpathy-Cramer, J., Gupta, R., and Raskar, R. Split learning for collaborative deep learning in healthcare, 2019. URL https://arxiv.org/abs/1912.12115.

Truong, N., Sun, K., Wang, S., Guitton, F., and Guo, Y. Privacy preservation in federated learning: An insightful survey from the GDPR perspective. *Computers & Security*, 110, 2021. URL https://doi.org/10.1016/j.cose.2021.102402.

Vepakomma, P., Gupta, O., Swedish, T., and Raskar, R. Split learning for health: Distributed deep learning without sharing raw patient data, 2018. URL https://arxiv.org/abs/1812.00564.

Wang, Y., Si, S., Li, D., Lukasik, M., Yu, F., Hsieh, C.-J., Dhillon, I. S., and Kumar, S. Two-stage LLM Fine-tuning with Less Specialization and More Generalization. In *ICLR*, 2024. URL https://openreview.net/forum?id=pCEgna6Qco.

Yang, G., Hu, E. J., Babuschkin, I., Sidor, S., Liu, X., Farhi, D., Ryder, N., Pachocki, J., Chen, W., and Gao, J. Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer, 2022. URL https://arxiv.org/abs/2203.03466.

Zhang, X., Zhao, J., and LeCun, Y. Character-level Convolutional Networks for Text Classification. In *NeurIPS*, 2015. URL https://proceedings.neurips.cc/paper_files/paper/2015/file/250cf8b51c773f3f8dc8b4be867a9a02-Paper.pdf.

A. Development of an Extensible Framework

We developed an extension of Flower (Beutel et al., 2022), one of the most widely used FL frameworks, to enable clients to request server functions. To facilitate its usage, the extension provides the same API as Flower. We here describe the implementation and provide some usage examples. The extension is available at `https://github.com/sands-lab/slower`, while a getting-started guide is provided at `https://github.com/BorisRado/slower_simple_examples`.

A.1. Architecture

In the framework, clients are seamlessly assigned a `server_model_proxy` attribute that, as displayed in Figure 3, allows them to invoke methods executed on the server. Similar to Flower, users need to implement the client logic, including the lower layers of the model and possibly the last layers in U-shaped architectures. Additionally, users must specify the logic for the server model in the `ServerModel` class. During training and evaluation, the `Client` object can invoke methods of the `server_model_proxy`, which are marshaled over the network using gRPC to the corresponding server model methods through the system's internal mechanisms. Furthermore, like Flower, the extension supports simulating the distributed algorithms on a single node by employing the `ray` library.

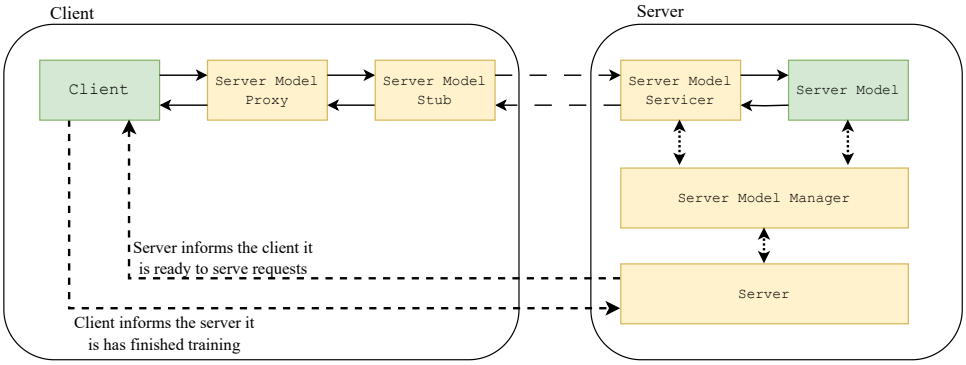

Figure 3. High-level architecture of the implemented framework in the case of gRPC deployments. Green boxes represent the training logic that the user needs to implement, while yellow boxes represent the framework's internal classes that are involved in the process of serving different types of client requests. Solid lines denote within-node communication while dashed lines denote communication over the network.

The server model proxy enables the client to invoke the server's logic either synchronously or asynchronously. In the synchronous case, the client waits for the server's response before continuing, as in the plain SL algorithm. In the asynchronous case, the client invokes the server logic and then continues its process, effectively enabling the two nodes to achieve overlapping computation. The difference between synchronous and asynchronous algorithms is visually represented in Figure 4.

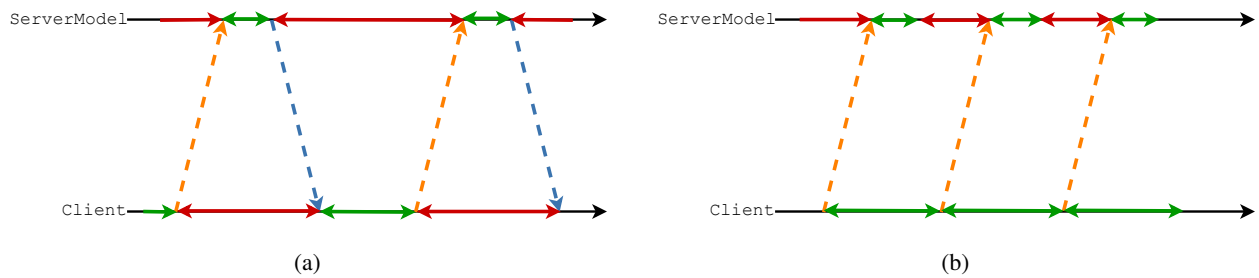

Figure 4. Visual representation of the workflow of (a) synchronous algorithms (e.g., plain SL and U-shaped SL) and (b) asynchronous algorithms with overlapping computation (e.g., FedGKT, FSL, and StreamSL). Dashed orange and blue lines denote client requests and server responses, respectively, while solid green and red lines indicate when the node is computing and when it is idle. In (b) the client is the bottleneck, hence the server needs to wait to receive client data.

To enable a great degree of flexibility, the client and the server can exchange arbitrary `numpy` data or raw `bytes`, hence allowing the user to apply any data serialization strategy.

A.2. Usage examples

We now provide some usage examples of the framework. The following code demonstrates how to implement the client for the Plain SL algorithm:

```python
class PlainSLClient(NumPyClient):  # NumPyClient is a class provided by the extension

    def fit(self, parameters, config):
        # parameters and config are equivalent to the corresponding arguments in Flower
        # prepare model and training dataloader

        for batch in trainloader:
            labels = batch.pop("labels")
            batch  = to_device(batch)  # possibly move data to GPU

            # forward pass on the client model
            embeddings = self.model(
                batch["input_ids"],
                batch["attention_mask"]
            )

            # compress embeddings with the lossless compression
            np_embeddings = compress_to_np_list(
                embeddings,
                batch["attention_mask"].sum(dim=1)  # number of tokens
            )

            # invoke the server model
            gradient = self.server_model_proxy.serve_gradient_update_request(
                embeddings=np_embeddings,
                labels=labels.numpy()
            )

            # uncompress the data
            gradient = expand_to_pt_tensor(gradient)[0]

            # backward pass
            self.optimizer.zero_grad()
            embeddings.backward(to_device(gradient))
            self.optimizer.step()

        # save model to disk or send parameters to the server
        return [], 0, {}
```

As mentioned, the library also allows invoking the server model asynchronously, effectively achieving overlapping computation. In the case of the StreamSL algorithm, the above training logic needs to be updated as follows:

```python
with torch.no_grad():
    # no need to store activations
    embeddings = ... # forward pass on the client as above

np_embeddings = ... # compress embeddings as above
self.server_model_proxy.serve_gradient_update_request(
    embeddings=np_embeddings,
    labels=labels.numpy(),
    blocking=False  # this makes the request asynchronous
)
# here client immediately continues
while self.server_model_proxy.get_pending_batches_count() > 20:
    # wait if the client is too much ahead of the server
    time.sleep(1)
```

In both cases, the user also needs to define the server model logic. For instance, for the Plain SL algorithm, the server model logic might look as follows:

```python
class PlainSLServerModel(NumPyServerModel):

    def configure_fit(self, parameters, config):
        ... # configure the server model

    def serve_gradient_update_request(self, embeddings, labels):
        # uncompress the data, set it to PyTorch format, and move it to GPU
        labels = torch.from_numpy(labels).to(self.device)
        embeddings, lens = expand_to_pt_tensor(embeddings)
        embeddings = to_device(embeddings)
        embeddings["hidden_states"].requires_grad_(True)

        # forward pass and loss computation
        predictions = self.model(**embeddings)
        loss = self.criterion(predictions, labels)

        # backward pass
        self.model.zero_grad()
        loss.backward()
        self.optimizer.step()

        # send to the client the gradient information
        gradient = compress_to_np_list(
            embeddings["hidden_states"].grad, lens
        )
        return gradient
```

The training logic for the StreamSL algorithm is very similar, the main difference being in the fact, that the server model can return `None` instead of the gradient, as such value is dropped by the framework.

Note, that in these examples we assumed to be defining the training logic in plain PyTorch, but as Flower, the extension is agnostic to the underlying deep learning library.

B. Further results

We here show the results when using BERT *base* as the upper bound model and to initialize the models in FedGKT and FSL. In Figure 5, we see that the main conclusions we made in Section 4.1 still hold, that is, models trained with FSL and FedGKT achieve comparable model qualities as standard training while training a shallow network purely on the client causes a significant drop in accuracy.

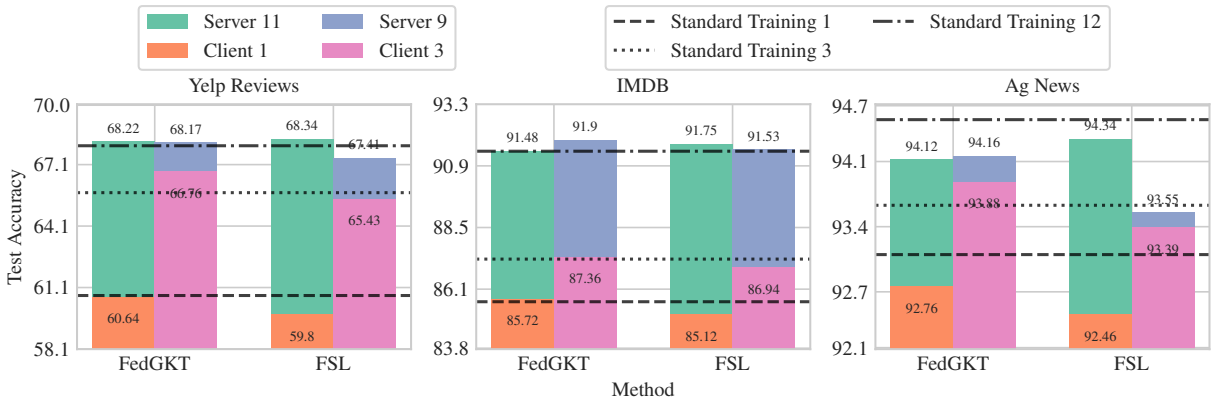

Figure 5. Test accuracy comparison between FedGKT and FSL relative to "Standard training X", where X is the number of layers in the model. "Node X" denotes the X-layer model trained on the respective node, e.g., Server 9 refers to the server model with 9 layers.