LoRA-SL: Low-Rank Adaptation for Continual Split Learning

Mateus C. Oliveira[1], **Heitor H. da Silva**[1], **Camilo H. M. dos Santos**[1], **Carlos Senna**[1], **Allan M. de Souza**[1], **Luiz F. Bittencourt**[1]

[1] Instituto de Computação (IC) – University of Campinas (UNICAMP)
Campinas – SP – Brazil

{m230437,h178291,c266738}@dac.unicamp.br

{crsenna, allanms, bit}@unicamp.br

Abstract. *In scenarios with drones on multiple missions and diverse tasks, it is common to use multiple servers training data models for different tasks. However, directly training a model on different tasks can lead to catastrophic forgetting, impairing model accuracy on old tasks and requiring retraining. Furthermore, it is important to preserve data privacy in these scenarios. To address this, we propose LoRA-SL, a Split Learning strategy that uses Low-Rank Adaptation (LoRA) to fine-tune client models rather than retraining them, while maintaining privacy. Experimental simulations performed with common classification datasets show that the proposed strategy allows clients to retain acquired knowledge while maintaining accuracy and reducing the number of training sessions.*

1. Introduction

5G communication environments rely on robust infrastructure and technologies to accommodate the ever-increasing number of connected devices that expect the services and applications they use to be reachable with a guarantee of quality and efficiency. In particular, with the growth of the Internet of Things (IoT) and its applications, there is a clear demand for reliable, adaptive infrastructure to support the massive number of these devices [Li et al. 2018].

In this context, machine learning (ML) offers many opportunities for creating systems capable of adapting to and predicting potential changes in the network environment, adjusting to better meet continuous demands [Fourati et al. 2021]. However, ML models may require significant computational capacity, making exclusively on-device training impractical for mobile and IoT devices with limited resources. To improve these limitations, Split Learning (SL) is an interesting ML option that can be used in a distributed manner, allowing multiple devices to collaboratively train models on local data with a server, without exposing that data or the models being trained. This collaboration can be dynamically employed by IoT devices on the network, allowing them to assist each other without sacrificing privacy.

Nevertheless, SL generally assumes that all participating devices are collaborating towards the same learning objective and that this ML task will not change throughout the lifespan of the devices. This brings disadvantages, especially in mobile devices with context-variant applications, such as drones and autonomous vehicles. In this scenario, the ability to progressively learn new tasks, one at a time, known as Continuous Learning,

is important [Aleixo et al. 2024]. For example, a service that provides a fleet of drones as data collectors and participating devices for SL workflows across different servers. Each drone collects data to train models in an SL workflow within a server's range, and each server has a different ML task using SL. Figure 1 shows this scenario with three servers, where the servers are respectively performing crowd size counting, traffic level recognition, and road condition recognition with the help of the drones.

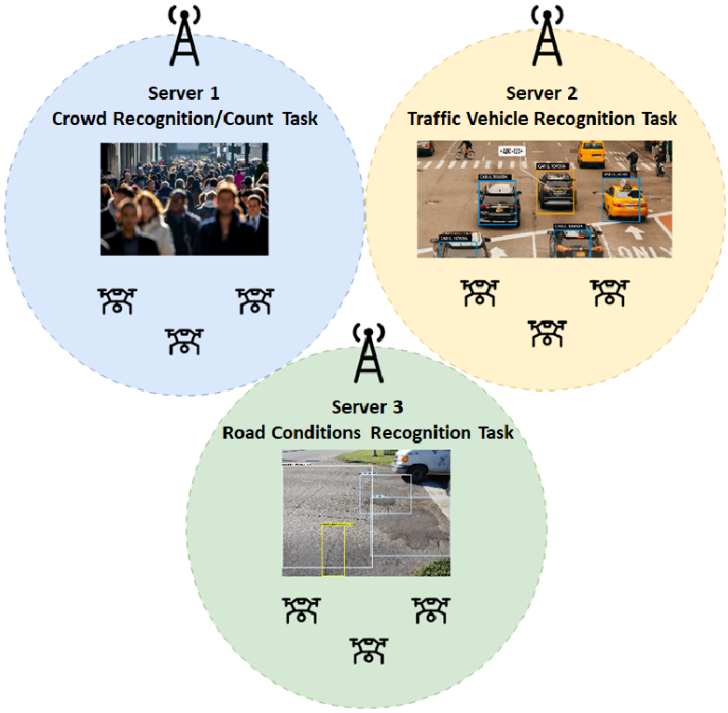

Figure 1. Proposed scenario: Drone devices participate in training models for ML tasks with SL, across different servers.

In this scenario, drones may require moving to another server to assist in training, which might have a completely different ML task in process. The participant thus has few options but to train their models from scratch on the new task or cache a model for each task; both choices can be quite resource-intensive. Furthermore, new training can worsen performance on previous tasks, as neural networks are susceptible to catastrophic forgetting [van de Ven et al. 2025], unlearning previously acquired knowledge when training for a different task.

With this in mind, we propose LoRA-SL, a Split Learning (SL) strategy augmented with the Low Rank Adaptation (LoRA) technique to mitigate knowledge loss when participants deal with "task mobility", that is, they need to learn multiple machine learning tasks. In our strategy, clients initially train a base model. For each new task, we train a task-specific parameter set instead of the base model. We use the LoRA technique, adapted for Convolutional Neural Networks (CNNs), to keep parameter sets substantially smaller than the base model in parameter count. Our tests show these parameter sets can store task knowledge, as participants maintain their accuracy across different task training and testing sessions.

We organized the remainder of this work as follows: Section 2 presents the fun-

damental concepts behind Split Learning, Continuous Learning, Catastrophic Forgetting, and LoRA. Section 3 reviews related work, while Section 4 presents the architecture and implementation of our solution. Section 5 discusses the results of experiments conducted on the implementation of our system. Finally, Section 6 concludes this work and offers possible future work.

2. Background

2.1. Split Learning

SL is a Distributed Machine Learning (DML) technique in which models are trained collaboratively by clients and a centralized server. Unlike other DML approaches, SL aims to allow models to cooperate in training while not only keeping training data private, but also keeping the trained model itself private between participants [Vepakomma et al. 2018].

To achieve this, the model is split in some way, separating it into a client-side and a server-side. In neural network settings, this split consists in selecting a layer in the network, and defining all layers before it as the client model, and all layers following it as the server model. The selected layer is referred to as the cut layer, and an example of it is shown in Figure 2.

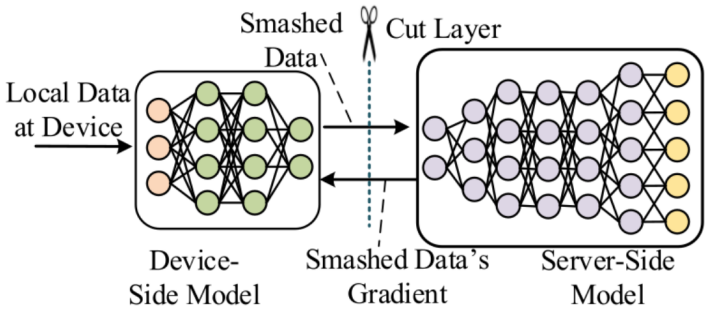

Figure 2. A model split in two under Split Learning. Source: [Wu et al. 2022]

With this separation in place, we perform client inference and training as normal, starting with the client model (which contains the input layer), but once the processing of the neural network reaches the cut layer, the activations produced at that point are sent to the server, which uses them to continue the forward propagation on its side. The outputs of the cut layer are referred to as "smashed data". The same process occurs during backwards propagation in training, with the server computing the gradients of the smashed data and sending them to the client so that it can update its own weights.

Therefore, in an SL workflow, the only data transferred between participants during inference is the smashed data, which, as the output of a single layer of the network, consists of a much smaller payload than a complete model or dataset. However, due to the model split, the client is unable to perform inference or training on its own. This increases the communication required between server and client, even if the payload is smaller [Hu et al. 2025]. Additionally, as the output of the split model lies on the server, the client must send the labels of its training dataset to the server for loss calculations to be performed, which can increase the payload and expose private information, but there are ways to avoid this, such as U-shaped SL [Lyu et al. 2023].

2.2. Continual Learning

Continual Learning refers to the ability of an intelligent system to be able to incrementally and continually learn from the data it receives, even as the distribution of the data, or the targeted learning task, changes [van de Ven et al. 2025]. This skill, prevalent in nature, has proven itself to be quite challenging for Artificial Intelligence to grasp, as the typical connectionist models (which include neural networks) struggle with retaining knowledge of previous tasks when learning a new one.

One of the causes of this struggle is a phenomenon referred to as catastrophic forgetting or catastrophic inference [McCloskey and Cohen 1989], where it was observed that sequentially training neural networks on disjointed data results in the models drastically losing performance on the tasks they were initially trained for, with even a small number of training rounds in new tasks being enough to disrupt the performance greatly. This idea can be visualized in Figure 3, where the accuracy in task 1 rapidly decays when training on task 2 starts.

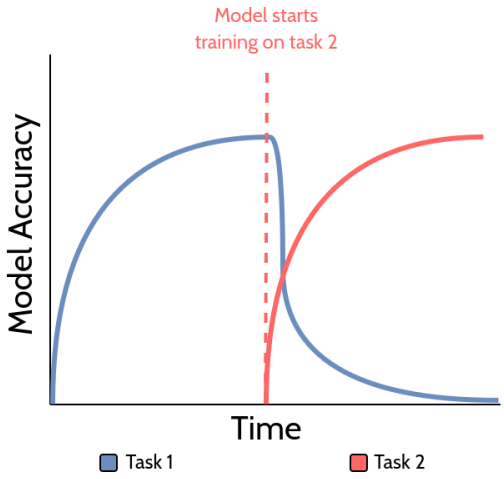

Figure 3. A model's performance over time, with two distinct learning tasks.

In order to combat catastrophic forgetting and achieve good performance in continual learning, ML systems must be adaptive and tolerant to fast changes in training scenarios, as well as being able to utilize the knowledge they have already acquired to act on tasks that have similarities to what they have already seen [Kudithipudi et al. 2022]. Topics such as Transfer Learning, model personalization and Knowledge Distillation emerged from these objectives, with the goal of permitting acquired knowledge to be perpetuated past individual tasks, models or learning scenarios.

2.3. Low Rank Adaptation and LoRA-C

LoRA is a model personalization technique, a kind of technique derived from Transfer Learning for adjusting ML model architectures with vast parameter counts. Initially developed for Transformers, particularly Large Language Models (LLMs), with the goal of adapting them to specific tasks in a more efficient manner, LoRA makes use of a technique called reparametrization to reduce the number of parameters that need to be retrained [Hu et al. 2021].

As opposed to traditional model fine-tuning, which retrains the entire LLM for a new application, reparametrization techniques involve freezing the parameters of the original LLM (such that they are no longer modifiable), and instead adjusting a separate set of parameters that will be combined with the frozen ones to achieve outputs with less error. In the case of LoRA, these separate parameters consist of a pair of matrices A and B, which are multiplied (BA) such that they can be added to the base frozen weights of the model. These two matrices have a rank of r, which is typically kept lower than the size of the input or output of the operation. This can be seen on the left-hand side of Figure 4.

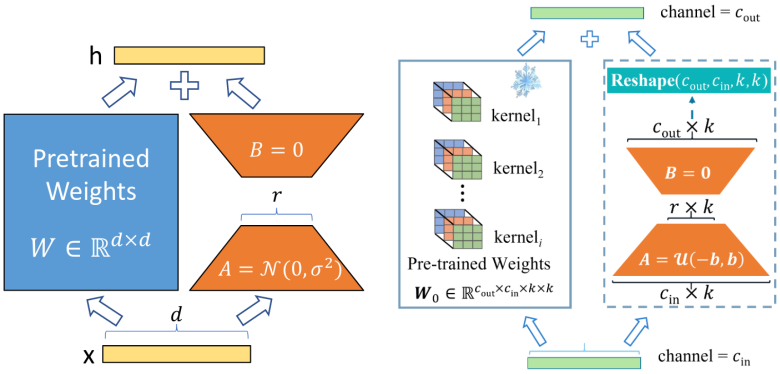

Figure 4. Reparametrization used in LoRA (left) and LoRA-C (right). Sources: [Hu et al. 2021], [Ding et al. 2024]

The main advantage brought upon by LoRA is that, by using a rank decomposition of two matrices, the total count of parameters can be much smaller than the base frozen weights, with the overall parameter count being controlled by r. In the case of LLMs, this allows for a base, pretrained model to be adjusted by manipulating only the two matrices, which is not only much less resource-intensive than a full retrain, but also opens the possibility to swap out the matrices depending on the desired application dynamically.

After the success of LoRA in fine-tuning LLMs, the underlying concept of the technology was applied to other ML architectures for the purpose of performing fine adjustments without requiring a retraining from scratch. LoRA-C (LoRA for CNNs) [Ding et al. 2024] is an implementation of LoRA for Convolutional Neural Networks, which are commonly used in Internet-of-Things (IoT) and embedded systems scenarios for tasks such as image classification and computer vision.

LoRA-C uses the same core concept of LoRA for LLMs, but with the convolutional layers of the CNNs being the target of reparametrization. This requires adjustments to the format and dimensions of the matrices, but the operations are the same: the kernels (the weights of the convolutional layer) are frozen, and the multiplication of two matrices is added to them prior to the convolution. Figure 4 shows the LoRA-C architecture for a single convolutional layer.

Similar to the use case in LLMs, LoRAs in CNNs allow for models to be adjusted in a less permanent and more resource efficient manner. Since CNNs are typically deployed in devices on the network edge, applying LoRAs could allow for these devices to more easily tailor their models to their local tasks, and possibly support continual learning to some degree. In this work, we verify this in an SL scenario in order for participating

clients to be able to learn multiple different tasks through LoRA after an initial base model has been trained and frozen.

3. Related Work

The integration of LoRA into distributed systems has been predominantly driven by the need to deploy LLMs on resource-constrained edge devices. Frameworks such as SplitLoRA [Lin et al. 2025a] and HSplitLoRA [Lin et al. 2025b] have successfully combined Split Learning with Federated Learning (Split-Fed) to parallelize training. However, their primary objective is computational efficiency: SplitLoRA focuses on reducing the communication payload during aggregation, while HSplitLoRA addresses device heterogeneity by dynamically adjusting LoRA ranks. Similarly, Memory-Efficient Split Federated Learning [Chen et al. 2025] utilizes LoRA to minimize server-side storage footprints during sequential training.

While these approaches optimize the mechanics of training, they largely overlook the dynamics of client adaptation. With the exception of FedALoRA [Yi et al. 2025], which introduces an adaptive aggregation mechanism for personalization, these frameworks assume a single-domain textual task. They do not address the challenge of multi-task learning, where a client model must adapt to entirely new tasks (e.g., shifting from one dataset to another) without undergoing catastrophic forgetting.

Research into LoRA for other model architectures such as CNNs remains sparse compared to Transformers. LoRA-C [Ding et al. 2024], introduced in Section 2.3, is a notable exception, which applies LoRA to CNNs on IoT devices. However, LoRA-C utilizes fine-tuning primarily to enhance model robustness against environmental data corruptions (such as noise or fog) rather than for learning distinct tasks, and crucially, does not consider distributed learning scenarios such as SL.

To address this gap in the literature, we propose the application of LoRA within SL specifically for multi-task adaptation. Unlike current state-of-the-art methods that focus on single-task LLM efficiency, our work explores how LoRA can enable a client CNN to transition between distinct visual domains applications while mitigating the catastrophic forgetting inherent in common model architectures.

Table 1. Comparison of Distributed LoRA Frameworks vs. Our Approach

Feature	**SplitLoRA** [Lin et al. 2025a]	**HSplitLoRA** [Lin et al. 2025b]	**Mem-Eff. SFL** [Chen et al. 2025]	**FedALoRA** [Yi et al. 2025]	**LoRA-C** [Ding et al. 2024]	**Ours**
Architecture	Split-Fed	Het. Split	Seq. Split-Fed	Federated	Cloud-Dev	**Split Learn**
Model	LLM	LLM	LLM	LLM	CNN	**CNN**
Goal	Efficiency	Heterogeneity	Memory	Personalization	Robustness	**Knowledge Acquisition, Payload Reduction, Memory Efficiency**
LoRA Role	Payload Reduct.	Resource Mgmt	Storage	Fusion	Robustness	**Fine-tuning**
Multi-Task?	Text Only	Text Only	Text Only	Non-IID	Corrupted	**Yes (Visual)**

Table 1 summarizes the above mentioned topics, highlighting the clear distinction between current research directions and our proposed approach. While existing frameworks have successfully optimized the efficiency of distributed training for LLMs, there remains a critical lack of empirical evidence regarding the adaptability of LoRA-based SL for different tasks.

A crucial part of our work will be to verify how effectively a client-side LoRA module can mitigate catastrophic forgetting when transitioning between tasks.

4. Methodology

4.1. LoRA-SL Architecture

The architecture of the proposed strategy consists of computationally-capable drones (clients) that can participate in SL workflows, and servers that have active model training tasks performed using those workflows. Each client has a ML base model and a LoRA cache, which is populated as the client communicates with and participates in tasks with different servers. As can be seen in Figure 5, each server is associated with a application task, that is, the training and usage of a split model capable of generating output for some application. Clients are responsible for collecting data, and using it for both training and inference with the server. The main objective of this strategy is to allow clients to quickly participate in differing ML tasks while mitigating knowledge loss in an fast and efficient manner.

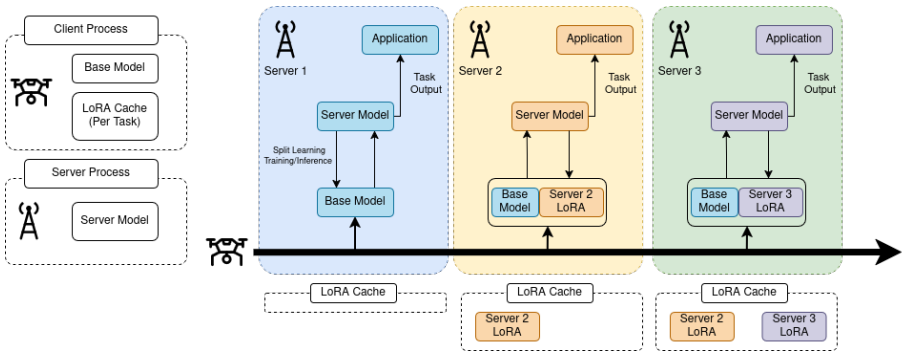

Figure 5. The LoRA-SL Architecture. The client process may move along servers, switching its current learning task accordingly.

Clients may move from one server to another to assist in different tasks. Clients initially train their base model on the first task they are assigned to, and, upon connecting to a different server, they create a LoRA parameter set for that specific server/task persisted in a client cache. This way, the base model can be kept frozen whenever the client is assigned to a different task from the one they started on. Knowledge of different tasks can thus be retained, either in the base model, for the initial task, or in instantiated LoRA parameters in the cache.

In order to validate this architecture, a minimal implementation was built using a test model and common ML classification tasks, which will be described in the following sections.

4.2. Model and Cut Points

The model used in our implementation is a deep CNN shown in Figure 6. The network is partitioned into two distinct segments: the **Client-Side** and the **Server-Side**. Three SL cut points were selected, which test the effects of keeping more layers, and layers that do not use LoRA, on the client process. As can be seen, the LoRA parameter sets are implemented in the convolutional layers of the CNN (`LoRAConv2D`); the weights of

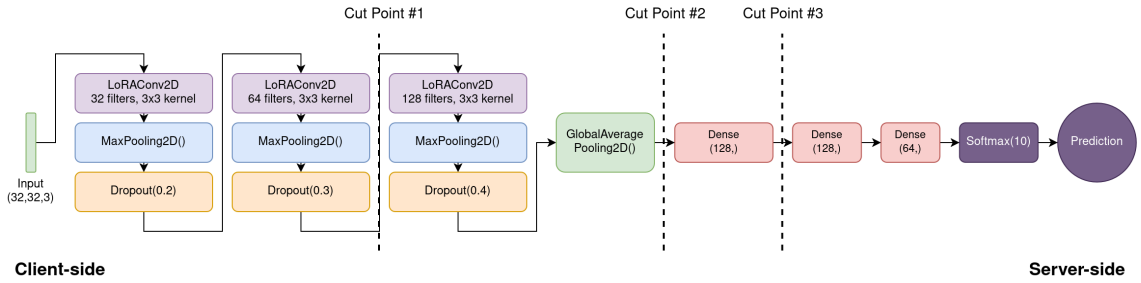

Figure 6. Proposed CNN and SL cut points.

these layers are frozen when LoRA is in use. Dense layers are kept unfrozen and trained normally on both client and server side.

The model expects input with the shape $32 \times 32 \times 3$, that is, 32×32 images in RGB format (3 channels). The output shape E_{dim} of the client's final layer varies according to the cut point, that being $6 \times 6 \times 64 = 2304$ in the first cut point, and 128 on both the second and third cut points. This is due to the first cut occuring between convolutional layers, while the other two after Pooling and Dense layers respectively.

4.3. Simulation Environment and Metrics

To validate our approach in a distributed setting, we developed a simulation environment in the Python programming language, using the TensorFlow library for the implementation of the ML models and the Ray framework[1] for instantiating clients and servers as separate, asynchronous processes.

For the purpose of emulating ML tasks in this environment, each server process is assigned a dataset from a pool of common classification datasets, which are detailed in Table 2. All datasets selected have 10 classification classes as output. As some of the datasets have samples that do not match the expected input dimension, all samples are padded (in case they are smaller than 32×32) or replicated (in case they only have one color channel). Notably, CIFAR-10 is a dataset that is notoriously hard for CNNs to learn with good accuracy [Lee 2025, Capanema et al. 2025], so while the expected accuracy is not guaranteed to be as high as the other datasets, it is expected to be maintained even after training on another dataset through LoRA.

Client-server interactions are abstracted through a message-based communication. Our simulated environment modeled clients moving from one task to another as a set of communication steps, with the client connecting to a server and receiving information on the task that is currently being trained. In order to test this task mobility specifically, we chose to simulate purely with processes; this abstraction freed us from simulating client movement to focus on the dynamic task switching instead. The remote procedure call interfaces used in our test environment are described below.

- `SendHello`, with which the client informs its presence to the server and receives the server's current task information.
- `SendTrainMessage`, in which the client sends a batch of activations and labels to the server for training (modifying the server-side model).

[1] https://github.com/ray-project/ray

- `SendTestingMessage`, in which the client sends a batch of activations and labels to the server for testing and validation (the server-side model is not modified).

In order to validate and measure the results of the proposed architecture, metrics pertaining both to the ML model as well as the goal of knowledge retention were chosen. In addition to the accuracy of the model, latency and epoch duration, the Forgetting Measure (FM) [Wang et al. 2024] metric will be used, which measures the average of the forgetting (the difference between the maximum accuracy achieved previously on a task, and its current accuracy) of all tasks prior to the current one. High FM values indicate that the accuracy of previous tasks has suffered, while values close to zero indicate that the model has retained its accuracy across different tasks.

Table 2. Datasets selected for implementation evaluation.

Dataset	Sample Shape	Training Sample Count	Test Sample Count
MNIST	$28 \times 28 \times 1$	60000	10000
Fashion MNIST (FMNIST)	$28 \times 28 \times 1$	60000	10000
CIFAR-10	$32 \times 32 \times 3$	50000	10000
SVHN	$32 \times 32 \times 3$	73257	26032
Kuzushiji MNIST (KMNIST)	$28 \times 28 \times 1$	60000	10000

5. Results

In this section, we present and analyze our experimental evaluation, designed to assess the effectiveness, scalability, and practical efficiency of the proposed architecture. The first experiment measures the impact of LoRA rank r and the selected cut points introduced in Section 4.2 on both the number of trainable parameters and the FM at the client side. We expect to determine whether increasing the rank parameter contributes to knowledge retention, and to identify the configurations that best balance model compactness and resistance to catastrophic forgetting. The second experiment focuses on scalability by measuring the average communication and training latency of multiple clients interacting with multiple servers. Finally, the third experiment evaluates training efficiency by comparing the time required for two clients (one using LoRA-SL and one without it) to sequentially traverse multiple servers while maintaining task accuracy above 80%.

5.1. Cut Layer and LoRA Rank Evaluation

The first experiment was performed by measuring the parameter count for each pair of cut point and LoRA rank, and then measuring the FM of a client training on a sequence of three tasks, using each cut point and LoRA rank combination. Alongside the three cut points, three values for the LoRA rank were tested based on the range of values originally tested with LoRA-C in [Ding et al. 2024], that is, $r \in \{32, 64, 128\}$).

Figure 7 shows the results of the parameter counting. The first cut point yielded a much lower LoRA parameter count overall, due to having the least number of layers on the client-side, but the number of parameters was more than half the size of the base model, shown as the red line, which is not very efficient. On the other hand, cut points 2 and 3 yielded comparatively smaller LoRA parameter sets, at roughly a third of the

parameter count of the base model; as such, a client using these cut points could store roughly LoRA sets for 3 different tasks in the same space it would take to store a separate task's model, when $r = 32$. It can also be seen that the rank increases the parameter count linearly, with the count doubling when the rank is doubled.

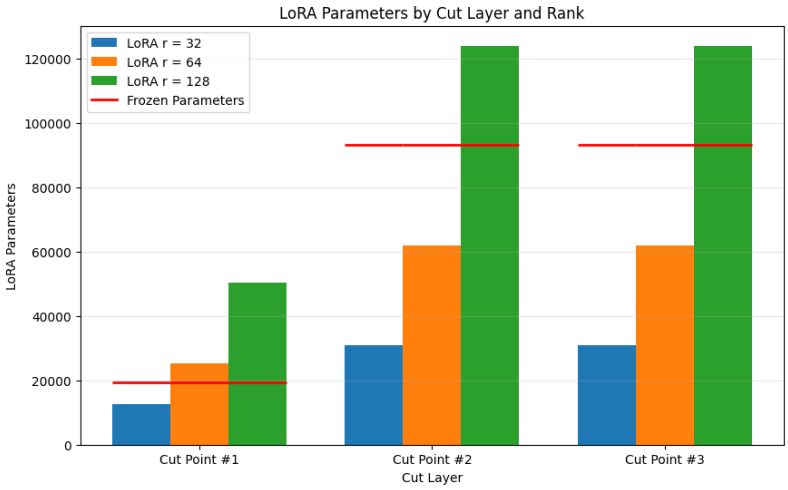

Figure 7. Number of trainable parameters in the LoRA, compared to the number of frozen parameters in the base model (red line).

Figure 8 reports the average FM across cut layers and LoRA ranks. Three datasets were randomly selected for training, and a client was trained on each one sequentially, with accuracy evaluated against all previous datasets after each training run. For each cut layer, five independent runs were performed, yielding 15 runs per LoRA rank and 45 runs in total.

FM is shown to be zero when the model is split at cut points #1 and #2, while cut point #3 shows variation regardless of LoRA rank. This occurs because the dense layer at cut point #3 does not include LoRA and is continuously retrained, leading to variations in accuracy in previous tasks after each training round in a new task. By contrast, cut points #1 and #2 place only LoRA-enabled layers on the client side, preserving accuracy throughout the training sequence.

These results show that both split location and rank affect the strategy's knowledge retention and memory efficiency. The dense layer at cut point #3 introduces variability that could, in principle, improve model generalization; however, no such benefit was observed. Instead, FM increased by approximately 0.076-0.086 (7.6-8.6 percentage points), indicating forgetting rather than generalization gain. It can thus be observed from this and from the parameter counts shown in Figure 7 that using cut point #2 and $r = 32$ results in a LoRA parameter set with the best knowledge retention and relatively lowest memory footprint (in terms of parameter count); this fact led these to be adopted as the default configuration in the following experiments, as the other configurations would either introduce forgetfulness into the model or have unnecessarily large LoRA parameter sizes.

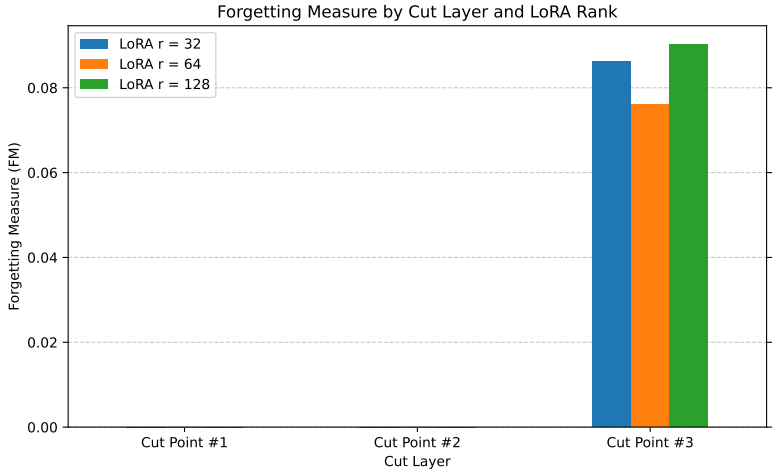

Figure 8. The forgetting measure (FM) of the different cut layers and LoRA ranks.

5.2. Latency Evaluation

The second experiment yielded results that are shown in Figure 9. The number of clients was increased until the maximum permitted in our environment, given the resources available in our test machine, with each client process being assigned an equal share of CPU and GPU resources.

All clients were made to follow the same "route", connecting to 3 different servers (and, as such, participating in 3 different tasks). Table 3 shows this route, where clients could stop at a server either to train for a number of epochs, or to test with a validation partition of the dataset. As can be seen, the growth of both latency and overall epoch duration (which includes not only the client-server communication interval, as well as the client-side processing and weight updates) was shown to be linear in relation to the number of clients.

#	Server Dataset/Task	Action	Epochs
1	SVHN	train	10
2	Fashion MNIST	train	10
3	KMNIST	train	10
4	Fashion MNIST	train	10
5	SVHN	train	10
6	SVHN	test	N/A
7	Fashion MNIST	test	N/A
8	KMNIST	test	N/A

Table 3. Route followed by clients.

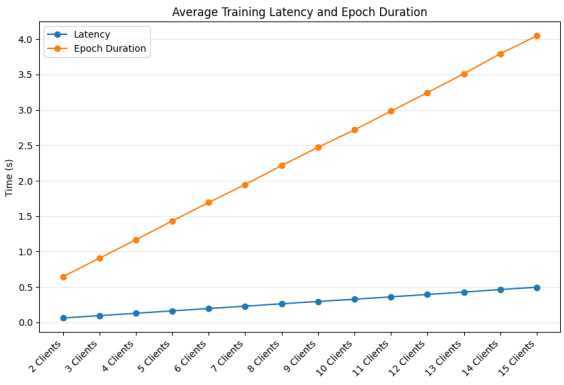

Figure 9. Average latency and epoch duration in relation to the number of active clients in the evaluation.

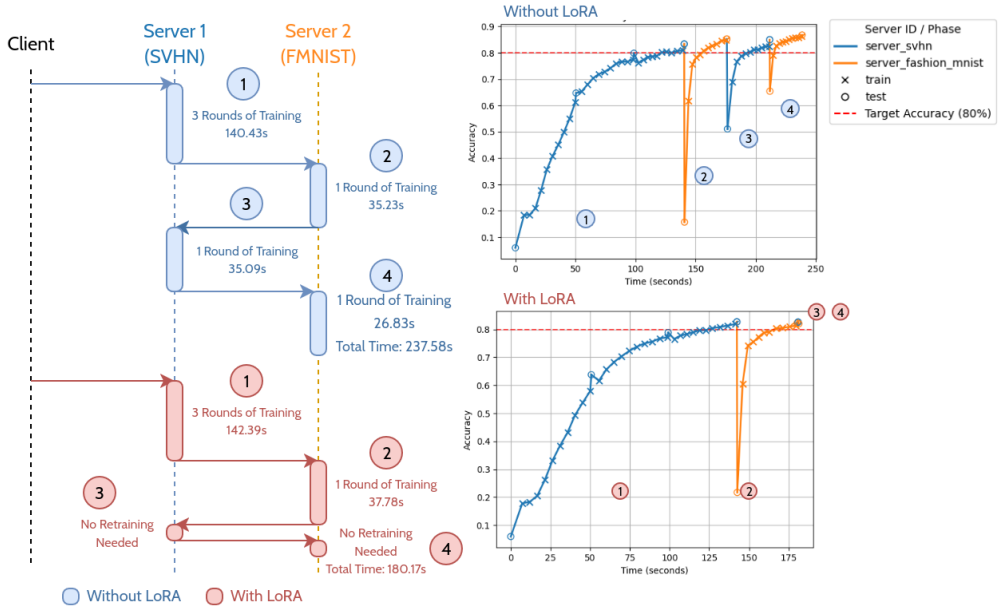

Figure 10. A sequence diagram for an experiment run, and a pair of graphs comparing the accuracy over time for a client without and with the usage of LoRA.

5.3. Convergence Time Evaluation

The third and final test compares the time taken for models to converge to a target accuracy and for this accuracy to be maintained over a series of different tasks. Clients with and without LoRA were made to traverse a number of servers in sequential order, training on 10-epoch long rounds until an accuracy of 80% was reached or 5 rounds had elapsed. After going through all servers, the clients would then go through them again, and the time it took for this 2-lap route to be completed was measured. To further exemplify this, Figure 10 shows the sequence diagram for an experiment run with two servers, as well as the corresponding results.

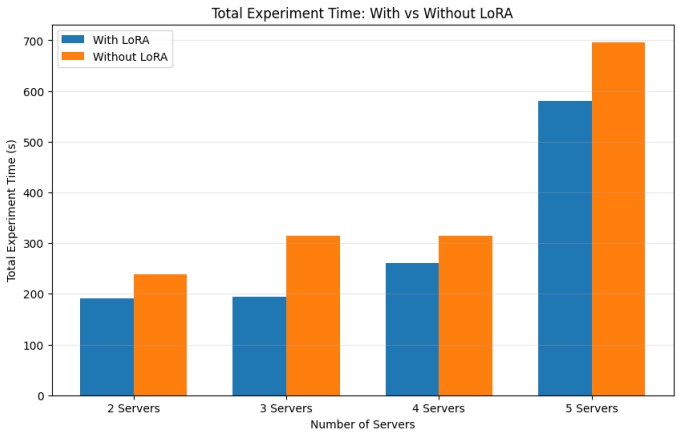

Figure 11. Total experiment duration as server count increases.

The time saved in the total experiment's duration is presented in Figure 11. The server tasks used were SVHN, Fashion MNIST, KMNIST, MNIST and CIFAR-10, in this

order. As can be seen in the graph, clients making use of LoRA were able to complete the experiment faster, as they were able to simply reload the parameter sets previously trained and achieve the same test accuracy as before. On the other hand, clients without LoRA were forced to retrain their models, as training on new tasks caused their models to forget the previous tasks. This shows there is a clear benefit in efficiency in the proposed scenario, as clients can reach the accuracy threshold more rapidly and perform inference, rather than needing to spend time retraining.

These experiments showed that there are benefits to the application of LoRA techniques in our proposed scenario. Not only are the LoRA parameter sets smaller than full models in parameter count, they allow for the drone clients to recall previous knowledge as they are assigned and train with different servers on different tasks. Even in situations where the base model and LoRA were unable to achieve the target accuracy, with the CIFAR-10 dataset, our strategy allowed for time to be saved in other tasks, leading to a faster execution.

6. Conclusion

Our work proposes a strategy for deploying multiple Split Learning workflows across multiple servers, with differing Machine Learning tasks, in a way that allows clients to quickly switch across servers and tasks while mitigating knowledge loss. We implemented the strategy in a simulated environment with asynchronous processes and interprocess communication, and validated this implementation through tests measuring the average accuracy, latency, forgetting measure (FM) and efficiency gain when making use of the strategy. The results showcased promising results as the usage of the strategy did, in fact, permit a retention of knowledge across tasks, with clients being able to achieve the same accuracy on tasks even after training on another.

While good results were demonstrated, more experiments and simulations are needed to better verify the performance. In future works, we aim to extend the implementation to a more full-fledged environment, where more realistic interaction between clients can be simulated. Furthermore, we aim to improve the SL model's capabilities and make it more robust, by exploring the addition of LoRA parameter sets on layers other than convolutional layers, as well as employing other continual learning techniques in conjunction with LoRA.

Acknowledgments

This work was supported by the São Paulo Research Foundation (FAPESP), grant 2021/00199-8 (CPE SMARTNESS), and process 2025/03795-1, process 2025/02093-3, and 2025/02185-5.

References

Aleixo, E. L., Colonna, J. G., Cristo, M., and Fernandes, E. (2024). Catastrophic forgetting in deep learning: A comprehensive taxonomy. *Journal of the Brazilian Computer Society*, 30.

Capanema, C. G. S., de Souza, A. M., da Costa, J. B. D., Silva, F. A., Villas, L. A., and Loureiro, A. A. F. (2025). A novel prediction technique for federated learning. *IEEE Transactions on Emerging Topics in Computing*, 13(1):5–21.

Chen, X., Li, L., Ji, F., and Wu, W. (2025). Memory-efficient split federated learning for llm fine-tuning on heterogeneous mobile devices.

Ding, C., Cao, X., Xie, J., Fan, L., Wang, S., and Lu, Z. (2024). Lora-c: Parameter-efficient fine-tuning of robust cnn for iot devices.

Fourati, H., Maaloul, R., and Chaari, L. (2021). A survey of 5G network systems: Challenges and machine learning approaches. *International Journal of Machine Learning and Cybernetics*, 12(2):385–431.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2021). LoRA: Low-Rank Adaptation of Large Language Models.

Hu, Z., Zhou, T., Wu, B., Chen, C., and Wang, Y. (2025). A Review and Experimental Evaluation on Split Learning. *Future Internet*, 17(2).

Kudithipudi, D. et al. (2022). Biological underpinnings for lifelong learning machines. *Nature Machine Intelligence*, 4(3):196–210.

Lee, C. H. (2025). Investigating CNNs performance on the CIFAR-10 dataset through hyperparameter tuning. *NHSJS Reports*. Received September 23, 2024; Accepted January 31, 2025.

Li, S., Xu, L. D., and Zhao, S. (2018). 5g internet of things: A survey. *Journal of Industrial Information Integration*, 10:1–9.

Lin, Z., Hu, X., Zhang, Y., Chen, Z., Fang, Z., Chen, X., Li, A., Vepakomma, P., and Gao, Y. (2025a). Splitlora: A split parameter-efficient fine-tuning framework for large language models.

Lin, Z., Zhang, Y., Chen, Z., Fang, Z., Chen, X., Vepakomma, P., Ni, W., Luo, J., and Gao, Y. (2025b). Hsplitlora: A heterogeneous split parameter-efficient fine-tuning framework for large language models.

Lyu, S., Lin, Z., Qu, G., Chen, X., Huang, X., and Li, P. (2023). Optimal Resource Allocation for U-Shaped Parallel Split Learning.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. volume 24 of *Psychology of Learning and Motivation*, pages 109–165. Academic Press.

van de Ven, G. M., Soures, N., and Kudithipudi, D. (2025). Continual Learning and Catastrophic Forgetting. pages 153–168.

Vepakomma, P., Gupta, O., Swedish, T., and Raskar, R. (2018). Split learning for health: Distributed deep learning without sharing raw patient data.

Wang, L., Zhang, X., Su, H., and Zhu, J. (2024). A Comprehensive Survey of Continual Learning: Theory, Method and Application. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(8):5362–5383.

Wu, W., Li, M., Qu, K., Zhou, C., Xuemin, Shen, Zhuang, W., Li, X., and Shi, W. (2022). Split Learning over Wireless Networks: Parallel Design and Resource Management.

Yi, X., Hu, C., Cai, B., Huang, H., Chen, Y., and Wang, K. (2025). Fedalora: Adaptive local lora aggregation for personalized federated learning in llm. *IEEE Internet of Things Journal*, pages 1–1.