Evaluating SplitFed Learning With Quantization

Kurtis Chow *Kenneth Pat* *Omar Samiullah*

Abstract

In a paper published in 2022, the CSIRO Data61 lab based in Australia proposed SplitFed Learning (SFL). This novel solution combines the characteristics of both federated learning and split learning to distribute model training and inference while keeping data private. While the paper discusses how they address privacy protection, it does not address combining this new distributed computing strategy with compression strategies to minimize latency. Our paper takes a step in this direction by naively introducing quantization of the smashed data and gradients to transmit smaller sets of data with lower precision data types aimed at maintaining high level of accuracies and lowering latency. We performed and obtained results from three versions of quantization on floating point operations using built-in PyTorch data types. Our results and analysis demonstrated that although performing quantization on SplitFed learning can minimize latency, it comes at a high price of losing accuracy. Our paper paves the way for further exploration in analysis on other data types and discussion on improving latency. We hope that this approach will increase interest in further development and testing of SplitFed Learning.

1 Introduction

The demands of modern machine learning models are ever-increasing. As a consequence, the field of data science has blossomed, driven by models that rely on vast amounts of data. A popular proposed solution is to access data from edge devices (e.g., smartphones, laptops) and leverage their processing power when idle. However, this approach raises unique privacy challenges. Two prominent methods have emerged to address these challenges: *federated learning* (FL) and *split learning* (SL).

Federated Learning (FL) is a distributed machine learning technique that utilizes a central server and a network of edge devices. In FL, clients only send their data to the model after it has been transformed, ensuring that data privacy is maintained. This approach contrasts with most distributed learning techniques that aim to parallelize computing machines to accelerate learning. Key features of FL include decentralized and heterogeneous data. Typically, FL has edge devices train local models on their own data samples, then exchange only the resulting parameters with the server. The server aggregates these parameters to build a global model, which is then redistributed to clients to repeat the cycle. This process continues until a specific condition is met, usually convergence of the model. Finally, clients are released, and the global model is finalized. Several challenges still exist in FL, such as device unavailability, minimizing potential harm to client devices, latency, and developing aggregation methods that do not require direct access to clients' data.

Split Learning (SL) is another distributed machine learning technique that aims to ensure the privacy of user data. This approach *splits* the model into a head and tail, assigned to the client and server respectively. The splitting point is usually referred to as the cut layer. The client trains the head model on their own data and only transmits the outputs at the cut layer to the server. The server takes outputs from several clients, aggregates them, and uses these parameters to complete the training of the tail model and essentially the whole model without ever accessing the actual data.

This process completes forward propagation, after which the server begins back-propagation until the cut layer. At the cut layer, the server sends its gradients to the clients, who finish back-propagation and then once more begins forward propagation. This cycle repeats until a condition is met, typically the convergence of some data.

SL offers several benefits, such as reduced computational cost and data transmission for clients, as they are restricted to the initial layers. In contrast, Federated Learning (FL) requires a more substantial network and has higher computational costs for client devices. However, SL still retains the benefits of FL in keeping data private.

SplitFed Learning (SFL) [4] combines SL and FL by sharing the computation load of training and inference between the server and client devices. SFL is built with a server that maintains the global model and performs aggregation, clients who train their own models on their own data, and a fed server located in proximity to the clients to handle heavy tasks like aggregating the client global model while minimizing the latency of distributing to the clients.

The server completes forward propagation on each client model using the cut layer's activations and then performs back-propagation on each. Afterwards, each client receives their respective activation gradients to complete back-propagation and finish forward propagation. Meanwhile, the server aggregates the results to update its model. This approach keeps all resources active and parallelized. A fed server was introduced to synchronize and aggregate client-side local updates for the clients' global model, providing an on-site server that can speed up performance for client devices to minimize their bottleneck.

We explored the effects of quantization on the data shared between the server and client. Quantization is a popular method to compress a large set of data into a smaller set

with minimal differences. It has gained popularity with the increasing size of large language models (LLMs) due to their vast number of parameters, ranging from thirteen billion to trillions.

This research builds on the SFLV1 framework and introduces quantization. By combining the SplitFed approach, which fuses the collaborative strength of federated learning and the computational efficiency of split learning, with model compression techniques such as quantization, we aim to explore the potential of applying compression to SFL. This combination should allow edge devices to collaboratively train and deploy sophisticated machine learning models while adhering to strict constraints on time, communication overhead, data privacy, and resource utilization.

2 Motivation

The rapid proliferation of edge devices in critical applications such as healthcare, medical diagnostics, and personalized patient monitoring has ushered in a new era of distributed intelligence. This technological advancement addresses the growing challenges of processing complex data in real-time while managing limited computational resources across diverse use cases. Traditional centralized approaches struggle to meet the unique demands of time-sensitive cases, for example, healthcare scenarios that require immediate analysis of patient data, remote monitoring, and rapid diagnostic decision-making.

A key focus of our study is to explore whether quantization (i.e., compression) can minimize latency while maintaining accuracy, thereby reducing the time required for training and inference. Achieving lower latency without sacrificing accuracy is particularly beneficial for a variety of time-sensitive tasks, where rapid decision-making is crucial.

The impact of this research extends beyond theoretical advancements. Distributed computing allows for real-time data analysis across various devices, leading to faster insights and responses. For example, machine learning models can be used with networks of sensors and wearable devices to provide immediate and accurate information. This capability can enhance early detection of problems, customized treatment plans, and ongoing monitoring. In specific areas, these systems can speed up analysis and reduce delays in decision-making. This framework aims to connect complex data processing with the need for quick and reliable solutions in various settings.

3 Related Work

When searching for related works, we found out that there are already several existing solutions (i.e. past approaches) that covered the topics of applying compression to split computing (SC) / split learning (SL) [3] and to federated learning (FL) [1]. These two strategies in past approaches are the high-level categories. However, our work focuses on applying compression to a novel approach introduced in a paper by Thapa *et al.* known as *SplitFed learning (SFL)* [4]. SplitFed learning merges the principles of SL and FL, while also eliminating their inherent drawbacks. When developing SFL, the authors considered the advantages of SL and FL, and also emphasized several topics, including data privacy and the model's robustness. Our approach is different from the aforementioned works because it addresses applying compression to SplitFed learning, which has its unique characteristics and advantages. As a result, since there is no existing work in literature that addresses the application of compression to SplitFed learning, we consider our strategy to be the major novelty of our project.

Past approaches in applying compression to federated learning (FL) focused on using compressed signals to reduce the overall communication overhead of solving the problem. Haddadpour *et al.* introduced a generalized version of local stochastic gradient descent (SGD) method for federated learning. The strategy used compressed message for up-link communication, thus reducing the overall size of the transmitted data. Additionally, it used a convex combination of the previous global model and the average of updated local models of users to create the new global model at the central node, allowing the overall model to achieve a faster time in convergence than state-of-the-art methods [1].

On the other hand, past approaches in applying compression to split computing (SC) addressed the shortcoming of existing split computing approaches. The application of compression to split computing introduced by Matsubara *et al.* focused on using a deterministic mapping to facilitate supervised compression. The supervised compression model is defined by introducing a representation vector to map a set of input data onto a set of targets. More importantly, supervised compression used its compressed features for predication tasks so that only relevant regions of a given image will have allocated bits (which correspond to pixels in the input image). As a result, the supervised compression model was able to achieve better compression rates compared to an input compression model (IC) [3].

Our project focuses on applying quantization as the main technique for the application of compression. Its main objective is to reduce computational and memory costs by transforming (mapping) the data in a machine learning model into new data types with lower precision. By doing so, we will be able to reduce the overall memory usage and computational costs.

4 Design

4.1 SplitFed Learning

This study is done by using the code provided from Thapa *et al.* [4], specifically the design for version 1 of Split-Federated

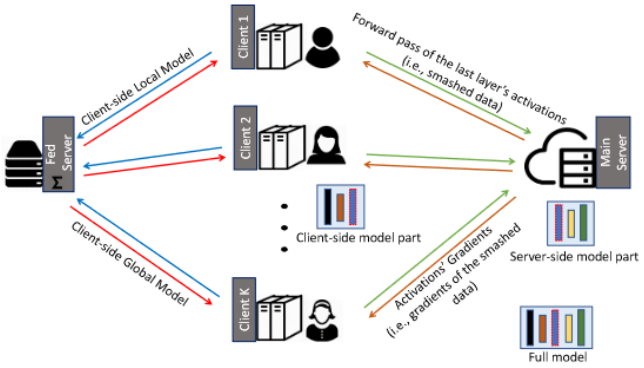

Figure 1: Overview of SFL from Thapa et. al [4]

Learning. This design has the server-side models of each client run in parallel each epoch. The code itself runs a simulation of a Split-Federated Learning where clients and the server are simulated as class objects / nodes.

The process for Split-Federated Learning begins at the fed server, a server located near the client devices for aggregating client-side local updates and updating the client-side global model. The global model is distributed across the clients who will then perform forward-propagation on this model using their local data. The resulting smashed data from the clients are then sent to the main server. Using the smashed data, the main server performs forward-propagation and then back-propagation for each smashed data in parallel. Next, the server aggregates the gradients from back-propagation to update the server-side model. Meanwhile, the clients receives its respective gradients from the server's back-propagation to complete its own back-propagation. The produced gradients from each client is then sent to the fed server who aggregates them to update the client-side global model and once more distributes this across the clients. This entails one epoch (round) and is repeated for however many times as chosen by the operator.

The fed server is introduced to provide a powerful machine for clients to delegate aggregating of the client-side updates to update the global client-side model. This provides an opportunity to minimize load on the clients while minimizing some latency issues if the main server was also required to update the client-side global model.

Aggregation is performed via federated averaging. Federated averaging (FedAvg) calculates a weighted average of the gradients for model updates. This has the benefit of allowing more than one batch update on local data providing some optimization improvement in contrast to similar methods such as federated stochastic gradient descent.

4.2 Quantization Methods

Due to time constraints of this project, we elected to use simple quantization methods of translating 32-bit floating point (`fp_32`) to 16-bit floating point (`fp_16`) and 16-bit brain floating point (`bfp_16`) made by Google Brain. These quantization methods were selected for their ease of implementation within the PyTorch library. `fp_32` was used as the baseline/control as it was default of the script during Chandra *et al.*'s paper [4].

4.3 Latency Design

We measured latency across the two passes between the clients and main server as seen in Figure 1(i.e. smashed data and activation's gradients). In the code, these are tensors that we saved using the PyTorch library tools to the disk. Then these are sent to a Raspberry Pi on the network using the `Paramiko` library. It's then downloaded back to the server device. The time taken to be downloaded to the Pi and then downloaded back onto the device is measured and treated as the latency. This method of measuring latency was selected because of ease of implementation and will produce times based off of the size of the tensors.

One issue was found during development where the Paramiko library's tools gave an IOError from a size mismatch between the real and locally allocated space for the tensor. A hot fix to the Paramiko library was done following a StackOverflow post [2]. While some issues did arise, we were able to successfully run each different quantization method for 200 rounds.

4.4 Analysis

Between each model, we measured the training and test accuracy of each round in addition to the average latency per pass of that round. Accuracies were obtained by splitting the dataset into training and validation on a 8:2 split. Analysis of these results was done using one-way ANOVA testing to determine if the differences were statistically significant and then graphing of the results. Graphs show both training and test accuracy per round of the given model. The average latency per round is shown upside down (i.e. inverse, closer to the top means it took less time) to better illustrate at what round was latency minimized and accuracy maximized.

5 Experiments

5.1 System and Specifications

Our experiment runs a simulation on a server located in the UCI IAS (Intelligent and Autonomous Systems Lab) at University of California, Irvine. The server runs Ubuntu 20.04.05 LTS (Jammy Jellyfish). It uses an x86_64 architecture, an NVIDIA RTX 2080Ti GPU equipped with 11 GB of GDDR 6 VRAM, an Intel Core i7-6700K as the CPU, and 64 GB of RAM.

Our code is modified from a version of the original code by Thapa *et al.* [4]. Their GitHub repository provides several variation of their codes, but the one version we chose comes

without using the socket and contains no DP or PixelDP. We selected the HAM10000 dataset from Kaggle, with a total size of 5.3 GB, because it was the default for the code. This dataset is a collection of 10,015 dermatoscopic images of common pigmented skin lesions. The images are labeled with what type of skin lesion and other labels. It was chosen as the code was already setup to run this dataset. From Thapa *et al.*, SFL had the most difficulty predicting this data set.

As for libraries, we mostly relied on PyTorch for training and quantization, and pandas for data processing, data wrangling, and data visualization. The SFL version is SFLV1 and is ran for 200 epochs (rounds). The local epochs at each client and loss rate are set to one and 0.0001 respectively to match the paper. Network layers are split at the third layer, after the 2D BatchNormalization layer. The code simulates a situation with five participating clients with one fed server and one main server.

The model runs a ResNet18 model architecture. This decision was due to being the default provided in the code. In the original paper [4], it achieved a 79% accuracy on the HAM10000 data set.

5.2 Graph Analysis

We performed our experiments on the dataset with three different versions: 32-bit floating point (`fp_32`, control group), 16-bit floating point (`fp_16`), and 16-bit brain floating point (`bfp_16`). We decided to choose them because these are the data types that are natively supported by the PyTorch library. The 32-bit floating point is represented in PyTorch as `torch.float32` or simply `torch.float`, and it is the default data type for floating point operations in PyTorch. As a result, it serves as the control group of our experiments. Similarly, the 16-bit floating point and 16-bit brain floating point are represented as `torch.float16` (or `torch.half`) and `torch.bfloat16`. Although we also discussed the possibility of performing quantization using 8-bit floating point, we decided to leave it as part of the future work because this data type has only "limited support" in PyTorch.

After collecting the accuracy and latency, we saved them to Excel spreadsheet files (`.xlsx`) and used the `matplotlib` library to plot the graphs, with the *x*-axis showing the number of rounds, and the two *y*-axes showing the accuracies (left) in percents (%) and latency (right) in seconds. Based on what we observed, there is no significant difference in training accuracies between each quantization method. This is the same between their test accuracies. The final training accuracy in each version always lies between 90% and 96%, whereas the final test accuracy always lies around 70%. Nonetheless, we do see that the test accuracy tends to fluctuate more over time in the `fp_16` and `bfp_16` versions.

On the other hand, the latency values in all three quantized versions fluctuated significantly over time. Latencies in the `bfp_16` version are much higher than the other two quantized

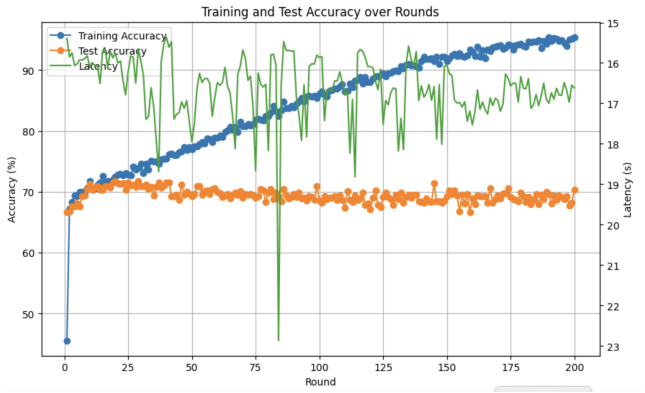

Figure 2: Accuracies with 32-bit floating point (`fp_32`)

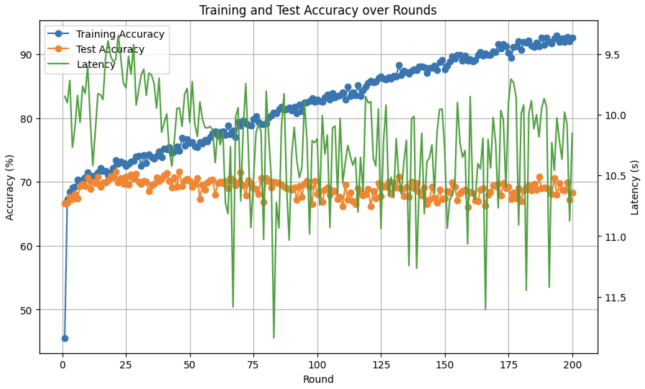

Figure 3: Accuracies with 16-bit floating point (`fp_16`)

versions. More importantly, in the `fp_32` and `bfp_16` versions, we observed two outliers in latency values. However, due to time constraints, we were not able to run each model a sufficient number of times to control or eliminate the outliers in latency.

5.3 ANOVA and Statistical Analysis

Statistical analysis of the results was done through one-way ANOVA testing. Results were statistically significant if the p-value was less than 0.05. Comparison across all quantization methods achieve p-values of 0.003, $4.30e^{-7}$, and 0 for training accuracy, test accuracy, and average latency respectively. Since all their values are less than 0.05, this indicates that the differences between groups were statistically significant.

Further testing was done between each group as pairs. Results can be found in Table 1. The difference between all methods were statistically significant on all metrics except for training accuracy between `bfp_16` and `fp_16`.

From our one-way ANOVA testing, we can conclude that the differences in results between all quantization methods are statistically significant. Therefore, we can compare them on their averages. Their averages for each metric across all

	Train Accuracy	Test Accuracy	Average Latency
All methods	0.0030	$4.305e^{-7}$	0.0
bfp_16 vs fp_16	0.9276	0.03816	$1.5715e^{-214}$
bfp_16 vs fp_32	0.0034	$6.2361e^{-8}$	$2.3435e^{-190}$
fp_16 vs fp_32	0.0044	0.0007	$1.9367e^{-281}$

Table 1: P-values between groups on each metric

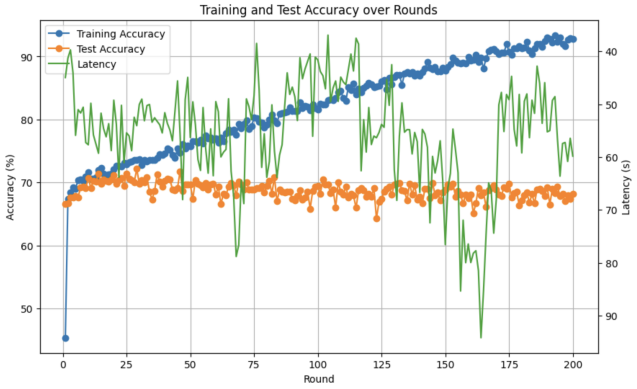

Figure 4: Accuracies with 16-bit brain floating point (bfp_16)

rounds can be found in Table 2. From these results, we can conclude that bfp_16 was the worst performing quantization method, having the highest latency and lowest accuracy, both training and test. fp_32, the control without quantization, had the highest accuracies but second highest latency. fp_16 only surpasses it in having the lowest latency. We can thereby conclude that while there is a statistically significant decrease in latency with fp_16, there is a statistically significant cost in accuracy.

6 Discussion

In this section, we discuss the implications of our findings regarding the performance of different numerical precision formats and the challenges encountered during our experiments.

6.1 Interpreting Results

We analyze the results obtained from our experiments with various quantization methods: bfp_16, fp_16, and fp_32. Understanding the reasons behind these results is crucial for drawing meaningful conclusions and guiding future research.

	Train Accuracy	Test Accuracy	Average Latency
bfp_16	82.048%	68.772%	55.429s
fp_16	82.116%	69.033%	10.218s
fp_32	84.409%	69.425%	16.622s

Table 2: Average for each quantization method on each metric

Interestingly, quantization does not necessarily compromise accuracy. While bfp_16 and fp_16 did have competitive raw accuracy levels compared to fp_32, ANOVA testing demonstrate that these lower precision formats result in a significant loss of model performance. bfp_16 exhibited poorer performance, potentially due to PyTorch's handling of data casting for this format, which may not be optimized for GPU execution. It is possible that bfp_16 is better suited for TPU architectures rather than GPUs.

While fp_32 consistently produced the highest accuracy among the formats tested, it ranked second highest in terms of latency. This suggests that higher precision formats enhance model accuracy but may introduce additional computational overhead affecting latency. Conversely, fp_16 experienced a loss in accuracy while showing lower latency compared to both bfp_16 and fp_32. This trade-off underscores the importance of balancing accuracy and efficiency based on specific application requirements.

6.2 Future Work

Future research could explore the impact of different datasets on SplitFed Learning and other quantization techniques. By utilizing diverse datasets, we can assess how various data characteristics influence model performance and communication efficiency while also pertaining to various applications. This exploration could lead to insights into optimizing the SplitFed Learning (SFL) framework for different applications, such as healthcare, defense systems, and search and rescue, where data privacy and real-time processing are critical.

Additionally, exploring advanced models such as transformers for vision tasks or natural language processing (NLP) could yield valuable insights into the effectiveness of quantization methods across various architectures. Investigating different models within a SplitFed Learning (SFL) framework would help determine whether quantization techniques are applicable and beneficial beyond the ResNet-18 architecture. Such research could significantly enhance our understanding of model robustness and adaptability in diverse contexts, ultimately contributing to the optimization of quantization

strategies in SFL.

6.3 Limitations

While our findings provide valuable insights into the performance of quantization methods, several limitations must be acknowledged:

1. **Latency and Bandwidth Constraints**: Significant fluctuations in latency were observed across different rounds, likely due to varying bandwidth between the server and Raspberry Pi clients. These bandwidth limitations may have impacted communication, leading to inconsistent latency measurements. Consequently, these factors complicated our ability to draw definitive conclusions regarding the efficiency of each quantization method.
2. **Time Constraints**: Due to time constraints, we were unable to conduct multiple tests across different configurations and scenarios, which may have affected our ability to fully understand the effects of quantization on model performance and latency. This limitation also hindered our capacity to confirm that bandwidth was the primary cause of the observed latency fluctuations. Future work should explore other compression techniques, models, and datasets to gain deeper insights into the applicability of quantization methods in SplitFed Learning (SFL).
3. **Optimization for Hardware**: The PyTorch library may not be fully optimized for GPU execution when using certain quantization methods like `bfp_16`. This could lead to suboptimal performance, as `bfp_16` may be better suited for TPU architectures or other ASICs. This limitation may have contributed to the poorer results observed with this format compared to others.
4. **Multiple Processes Running**: The presence of multiple machine learning processes running on the server during our experiments may have compromised results by introducing additional load and competition for resources.

These limitations highlight the need for careful consideration when interpreting our results and suggest that further investigation is necessary to fully understand the dynamics at play in distributed computing environments.

While our study demonstrates mediocre results with quantization methods in a SplitFed Learning context, it emphasizes the importance of addressing network-related issues and exploring diverse datasets and model architectures in future research.

7 Conclusion

In conclusion, our experiments lied on the intersection between the fields of SplitFed learning and quantization. We leveraged the power of PyTorch to perform three versions of quantization on top of the original SplitFed learning algorithm by Thapa *et al.* [4], by trying to combine the advantages and characteristics from both SplitFed learning and quantization. We were fortunate to be able to utilize the computational power of the server at UCI IAS for our experiments, since the major portions of our code rely on the CUDA interface, which is supported only in NVIDIA GPUs. Our results and the analysis from the one-way ANOVA testing revealed that although the difference is small, there is a statistically significant difference in almost all metrics, implying that quantization would sacrifice the accuracy for a lower latency. On the other hand, we determined that the `bfp_16` has a significant higher level of latency, indicating that it might not be an optimal choice of data types for quantization-related operations. In the end, the topics in our paper remain wide open for further research. We recommend future researchers to further explore other types of datasets, other data types for representation in performing quantization with SplitFed learning, as well as methods for improving the overall latency in quantization.

References

[1] Farzin Haddadpour, Mohammad Mahdi Kamani, Aryan Mokhtari, and Mehrdad Mahdavi. Federated learning with compression: Unified analysis and sharp guarantees. In Arindam Banerjee and Kenji Fukumizu, editors, *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 2350–2358. PMLR, 13–15 Apr 2021.

[2] joshv2. "ioerror: size mismatch in get!" when retrieving files via sftp, Dec 2018.

[3] Yoshitomo Matsubara, Ruihan Yang, Marco Levorato, and Stephan Mandt. Sc2 benchmark: Supervised compression for split computing, 2023.

[4] Chandra Thapa, Mahawaga Arachchige Pathum Chamikara, and Seyit Camtepe. Splitfed: When federated learning meets split learning. *CoRR*, abs/2004.12088, 2020.