Communication-Efficient LLM Training for Federated Learning

Arian Raje

May 2024

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Virginia Smith, Chair
Zhihao Jia
Gauri Joshi

Submitted in partial fulfillment of the requirements for the Master's degree in Computer Science.

Keywords: Federated Learning, Sparsity, Efficiency, LLMs

Abstract

Federated learning (FL) is a recent model training paradigm in which client devices collaboratively train a model without ever aggregating their data. Crucially, this scheme offers potential privacy and security benefits for users by only ever communicating updates to the model weights to a central server as opposed to tradition machine learning (ML) training which directly communicates and aggregates data. However, FL training suffers from statistical heterogeneity as clients may have differing distributions of local data. Large language models (LLMs) offer a potential solution to this issue of heterogeneity given that they have consistently been shown to be able to learn on vasts amount of noisy data. While LLMs are a promising development for resolving the consistent issue of non-I.I.D clients in federated settings, they exacerbate two other bottlenecks in FL: limited local compute and expensive communication. This thesis aims to develop efficient training methods for LLMs in FL. To this we employ two critical techniques in enabling efficient training. First, we use low-rank adaptation (LoRA) to reduce the computational load of local model training. Second, we communicate sparse updates throughout to training to significantly cut down on communication costs. Taken together, our method reduces communication costs by up to 10x over vanilla LoRA and up to 5x over more complex sparse LoRA baselines while achieving greater utility. We emphasize the importance of carefully applying sparsity and picking effective rank and sparsity configurations for federated LLM training.

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increased ubiquity of edge devices, such as IoT devices and smartphones, has made federated learning (FL) paradigms a feasible way to train machine learning (ML) models [1]. In traditional ML applications, a central server would first aggregate the data from disparate sources. A model would then be trained at the server-level using this aggregated data. While FL applications similarly involve a central server and disparate data sources, which we refer to as "clients", FL offers an alternative to traditional training procedures. FL applications will instead first distribute the model parameters from the central server to the clients (download phase). The clients then independently train the model parameters they receive from the central server on their local data. Once local training is complete, the clients send the model parameters back to the central server (upload phase) where the central server aggregates the clients' models into a new global model. In total, the above steps constitute a single "communication round" for the training procedure. This process is repeated for multiple communication rounds, with each round involving sampling clients, training local models, and aggregating the models at the central server. In this training scheme, data never leaves the clients' devices and only the model parameters are communicated between the central server and the clients. As a consequence, FL offers potential privacy benefits over traditional ML [2]. Additionally, because models are trained at the client-level, recent FL studies even suggest that FL may avoid the sustainability issues of distributed data center ML [3]. FL-based training schemes are particularly useful in large networks of similar devices and have already been productionized at scale [4, 5].

While FL continues to become more valuable in practice, practical concerns about its utility remain. In particular, the use of Large Language Models (LLMs) has become standard for a vast number of ML problems [6]. In many settings, the LLM being used may have billions of trainable parameters. The scale of these LLMs presents serious compute and communication bottlenecks for distributed training schemes. Since the advent of LLMs, a common strategy to use LLMs has been the pretrain-then-fine-tune framework. In essence, LLMs like GPT or BERT, which have already been pretrained on a large corpus of data, can be fine-tuned on a downstream task. A consequence of this framework is that the updates to the pretrained model weights take on low-rank structures, eliminating the need for full fine-tuning of all the model weights. In recent years, adapter methods for LLMs have been developed to incorporate these ideas by injecting a small set of trainable parameters into each transformer block and freezing the remaining parame-

ters of the model [7, 8]. Therefore, when a pretrained LLM is being fine-tuned on a downstream task, only a smaller portion of parameters must be trained.

Adapter methods offer a concrete way to reduce the compute and communication loads of LLM training in a distributed or federated setting. However, empirically, adapters themselves need to be quite large to retain the original model's utility. Training an LLM in a federated setting would still require communicating massive updates at each round. While adapter methods reduce some of these communication bottlenecks, in practice they can still lead to slow communication while seeing more significant drops in model utility. They may additionally increase storage and inference costs by increasing the number of total parameters included in the model [9].

LLMs broadly have achieved state-of-the-art performance in multiple domains and in many ways have become the standard approach for modeling with large amounts of data. Nonetheless, they remain difficult to implement in FL. The fact that FL applications involve communication over a wireless network means that coordinating large model updates from a large array of clients is specifically difficult and has been a primary concern in utilizing LLMs for FL. **The goal of this thesis is to introduce a method to perform communication-efficient LLM training for FL.** To this end, we introduce **F**ederated **Lo**RA with **S**imple **S**parsity (FLoSS). We specifically employ sparsity to low-rank adaptation (LoRA) during only the download and upload phases of FL training in order to retain the model's utility while restricting communication. We additionally show that given an arbitrary communication budget, we can accurately select a rank and download/upload sparsity ratio to perform accurate model training without ever exceeding the budget. We summarize our contributions as follows -

1. To the best of our knowledge, we are the first to apply unstructured sparsity to LoRA for efficient federated fine-tuning. We focus on unstructured (weight-level) sparsity because it has been shown to outperform structured (block-level) sparsity in centralized settings.
2. We propose FLoSS, a simple baseline that applies a constant top-k sparsity only to communication. This method can reduce communication costs up to 10× while matching the performance of dense LoRA on several FL image and text tasks.
3. We simulate an FL training procedure on a network with realistic download and upload communication speeds. Given a communication budget, we demonstrate ways to accurately select a LoRA rank and download/upload sparsity ratios that maximize the model's utility within that budget.

FLoSS aims to make LLM training more feasible in a federated setting. LLM training in resource-constrained environments remains an open research problem and an important field of study as LLMs grow in size. We hope to contribute to this growing field of work by proposing methods to improve efficiency while retaining utility in real-world settings.

Chapter 2

Background

2.1 Federated Learning Methods

2.1.1 Problem Statement

FL applications train an ML model across a distributed network of clients. Traditional FL applications involve a central server and a set of k client devices. At each communication round, a sample of the k devices each download a copy of the global model weights, train the model on a local objective, and upload the locally trained model weights back to the central server. The communication round ends with the central server aggregating the weights into a new global model. The process is depicted below in Figure 1. The objective of an FL application is traditionally described as follows where the model is parameterized by weights w -

$$\min_{w} F(w), \text{ where } F(w) := \sum_{i=1}^{k} p_i F_i(w) \tag{2.1}$$

F_i describes the local objective for client i. In our setup, we treat F_i as the loss of the model parameterized by w with respect to the local data on client device i -

$$F_i(w) = \frac{1}{m_i} \sum_{j=1}^{m_i} l_i(x^{(j)}, y^{(j)}; w) \tag{2.2}$$

Each client has m_i local examples and local loss function l_i. Our global objective F is weighted by parameters p_i where each $p_i \geq 0$ and $\sum_{i=0}^{k} p_i = 1$. While multiple weighting schemes for clients exist, we define $p_i = \frac{1}{k} \;\; \forall i \in [1 \dots k]$. With this weighting, the global objective function treats each client equally regardless of the distribution of local data. Therefore, an FL training procedure aims to find parameters w that minimize the average of the loss across the k clients. We note that there exist certain FL settings that train multiple models (optimizing multiple w_i's) or have different tasks for different clients (using different l_i's). Our focus is single-task, single-model FL where each client uses the same loss function and optimizes a single set of parameters w. The next section described two common procedures used to achieve this objective, namely FedAvg and FedAdam.

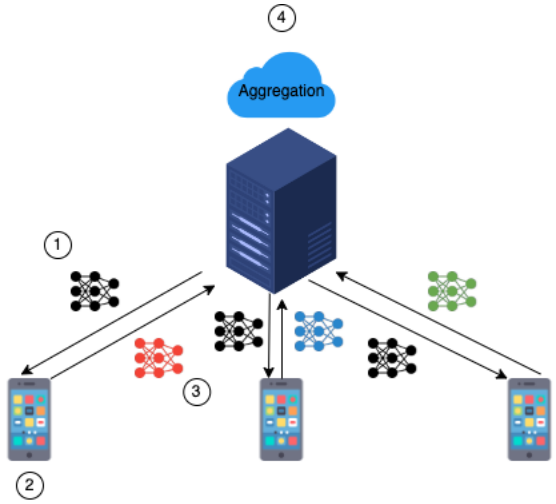


Figure 2.1: FL procedures involve 4 steps in each communication round. (1) Clients download model weights from the central server. (2) Clients train the model on local data. (3) Clients upload local model weights to the central server. (4) The central server aggregates client updates into a new global model.

2.1.2 Optimization Techniques

We present two algorithms that aim to find model parameters w that minimize the objective F. These methods draw from historical works in distributed optimization. The first method is Federated Averaging (FedAvg). Consider a total of k client devices. At communication round t, n clients are sampled from the k total clients. These n clients each download a copy of the global model with parameters w_t from the central server and train the model for e epochs on local client data. This results in local models $w_t^{(1)}, w_t^{(2)}, \ldots, w_t^{(n)}$. These models are uploaded back to the central server. The central server then aggregates by averaging these models to define the new global model. This update is formulated as follows -

$$w_{t+1} \leftarrow \frac{1}{n} \sum_{i=1}^{n} w_t^{(i)} \tag{2.3}$$

This process is repeated for a total T communication rounds. In this straightforward method, successive averaging of client updates aims to make the final global model w_T accurate for all clients k despite sampling a fraction of the clients in each communication round. In settings with large k, it is possible for $n << K$ to still result in an accurate final model as has been shown empirically in many studies. Federated Adam (FedAdam) proceeds similarly to FedAvg with a slight adjustment made to the aggregation mechanism. In FedAdam, once the clients performing local training to calculate $w_t^{(1)}, w_t^{(2)}, \ldots, w_t^{(n)}$, they each calculate $\Delta_t^{(i)} = w_t^{(i)} - w_t$ and upload $\Delta_t^{(i)}$ back to the central server. The central server calculates the averages of these differences $\Delta_t = \frac{1}{n} \sum_{1=1}^{n} \Delta_t^{(i)}$ and uses this Δ_t as a "pseudo-gradient" for an Adam optimizer. In other

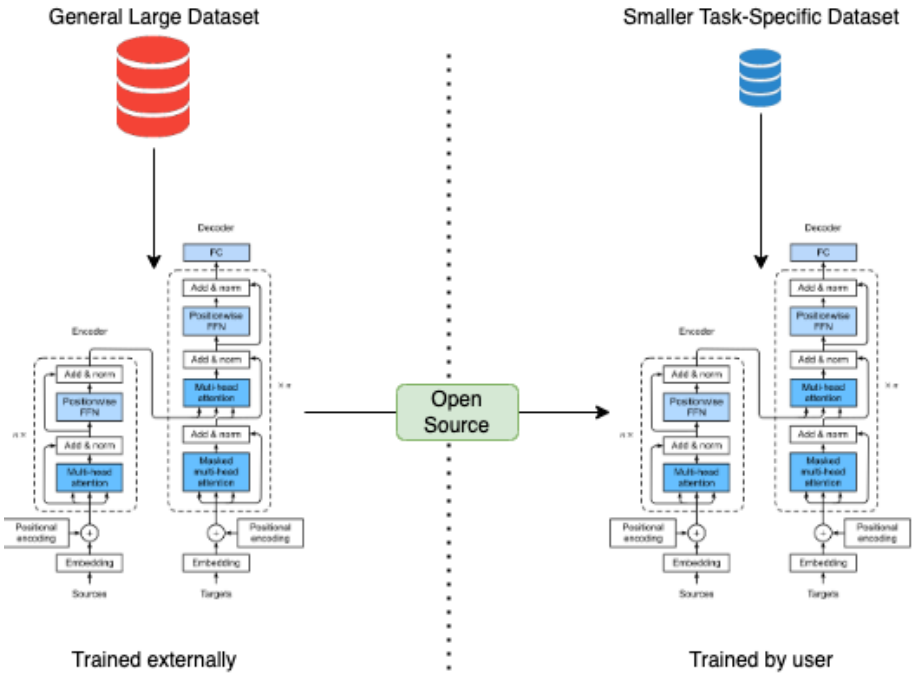


Figure 2.2: LLM fine-tuning involves taking an open-source model trained on generic data and adapting the weights on task-specific data.

words the update can be formulated as -

$$\begin{aligned} m_{t+1} &\leftarrow \beta_1 m_t + (1-\beta_1)\Delta_t \\ v_{t+1} &\leftarrow \beta_2 v_t + (1-\beta_2)\Delta_t^2 \\ w_{t+1} &\leftarrow w_t + \gamma \frac{m_t}{\sqrt{v_t}+\epsilon} \end{aligned} \tag{2.4}$$

FedAdam has been shown to outperform FedAvg in a variety of settings. For this reason, we utilize the FedAdam optimization method for the remainder of our experiments. We aim to benchmark FLoSS against the best possible baseline. Therefore, using FedAdam was the natural choice to compare against for our communication-efficient method for federated training.

2.2 Large Language Model Fine-Tuning

LLMs have become a standard approach for a variety of ML problems due to their state-of-the-art performance in a broad host of domains (e.g. language translation, question answering, next-word prediction, large-scale vision, etc.). An important finding is that the performance of LLMs scales well with respect to the number of parameters in the model. This runs contrary to other architectures like CNNs, RNNs, and LSTMs where performance on a task reaches a saturation point despite an increase in the number of parameters. However, as LLMs continue to grow in size, they become increasingly hard to train efficiently. Coupled with the fact that LLMs require inordinate amounts of data in order to achieve their state-of-the-art accuracy, training

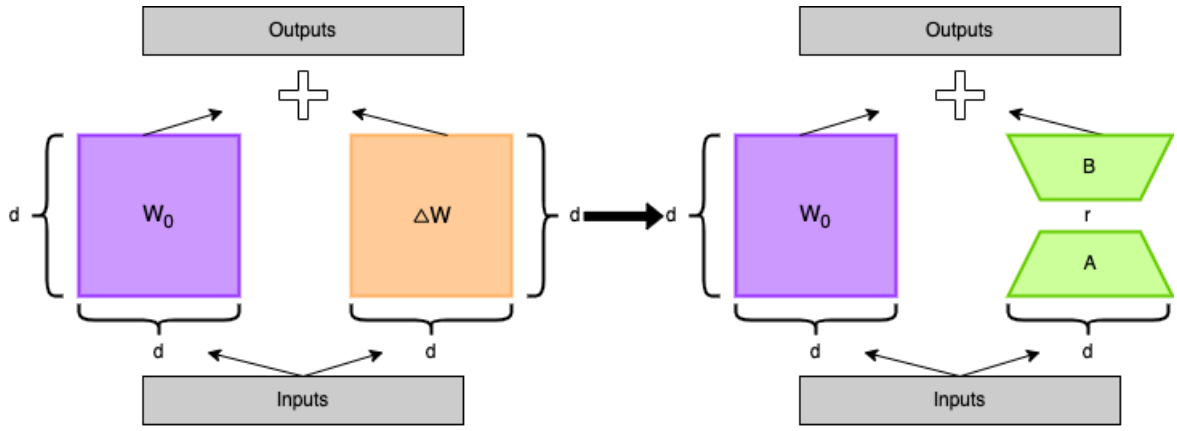


Figure 2.3: LoRA inserts small trainable matrices into the model architecture and freezes the remaining model weights.

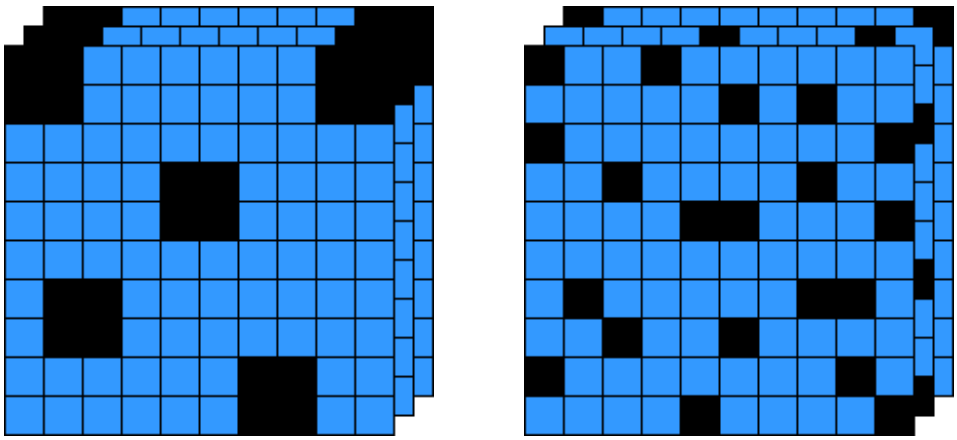
LLMs presents serious logistical and computational challenges. To address some of these challenges, recent literature has focused on a pretrain-then-fine-tune setup for LLM training. The core idea is to use open-access LLMs that have been pretrained on a large corpus of public data and then fine-tune the model weights on a domain-specific task. Models like LLaMA, GPT, BERT, and ViT have become standard open-source models to use in this training paradigm for tasks including representation learning, chat, and image classification. While this pretrain-then-fine-tune setup helps resolve some of the massive data requirements for LLM training, it still suffers from the computational, memory, and storage requirements for training all the weights of an LLM.

Adapter methods improve the computational efficiency of LLM training by reparameterizing the updates to the model. Instead of training all the weights of the LLM, adapter methods inject a small set of trainable weights into the model architecture and freeze the original model weights at their pretrained value. These methods are inspired by the idea that the change in weights from a pretrained model to a fine-tuned model exist in a low-rank space. The most frequently used adapter is Low-Rank Adaptation (LoRA). LoRA reparameterizes weight updates as follows. Consider an initial weight matrix $W_0 \in \mathbb{R}^{d\times d}$. The update to W_0, which we call $\Delta W \in \mathbb{R}^{d\times d}$ can be defined as a product BA where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times d}$. In this case r is a hyperparameter and is generally defined in a way such that $r << d$. In order to make training using LoRA more efficient, W_0 is frozen at its pretrained value and only B and A receive gradient updates. The forward pass of the model with input x can simply be defined as follows -

$$Wx = (W_0 + BA)x = W_0x + BAx \tag{2.5}$$

B is initialized as the 0 matrix while A is initialized as $N(0, \sigma^2)$ so at the initialization of the LoRA parameters, $Wx = (W_0 + 0A)x = W_0x$. Ultimately, by training the LoRA parameters instead of W_0 directly, we train only $2dr$ parameters as opposed to d^2 parameters. Choosing an adequately small r can greatly improve the efficiency of fine-tuning.

Inserting LoRA parameters into an LLM is straightforward. In each transformer block of an LLM



(a) Structured Pruning (b) Unstructured Pruning

Figure 2.4: Structured pruning sets entire structures to 0 while unstructured pruning sets individual weights to 0.

there is a Multi-head Self-Attention (MSA) mechanism with weight matrices $W_Q, W_K, W_V \in \mathbb{R}^{d\times d}$ where d is the embedding dimension. With input X, we define $Q = XW_Q$, $K = XW_K$, and $V = XW_V$. MSA is then calculated as -

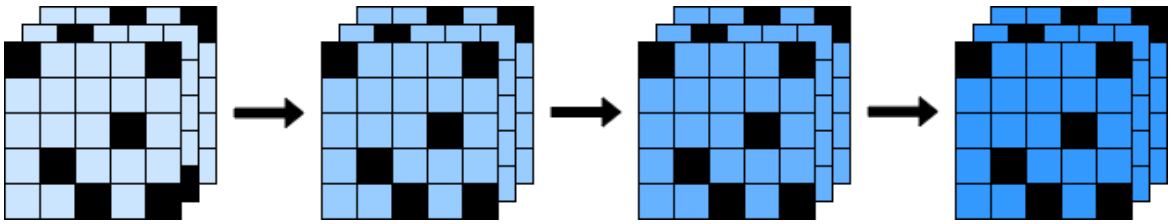
$$\text{MSA}(X) = \text{softmax}(\frac{QK^T}{\sqrt{d}})V \tag{2.6}$$

In this calculation, W_Q, W_K, and W_V are all trainable parameters. With LoRA, we insert B_Q, A_Q, B_K, A_K, and B_V, A_V and freeze W_Q, W_K, and W_V at their pretrained weights. We use LoRA for our experiments for the following reasons -

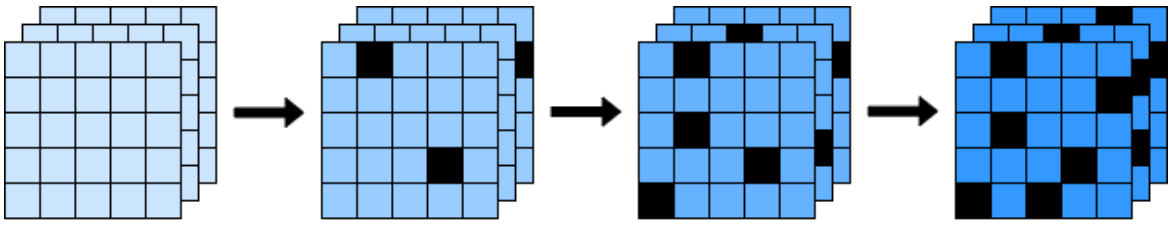
- LoRA parameters can be easily merged back into the model by calculating $W = W_0 + BA$ to reduce inference and storage costs.
- LoRA can be easily integrated with any transformer-based architecture as these architectures all use some for MSA and LoRA parameters can be defined for the weight matrices used in MSA.
- Using LoRA significantly reduces VRAM consumption making local LLM training feasible for client devices in an FL setting.

2.3 Sparsity and Model Compression

Model compression techniques aim to reduce the size of an ML model by altering the weights or structure of the model. These methods have been used for various purposes including improving the training efficiency, decreasing the storage requirements, and reducing the inference latency of ML models. While there exist multiple methods for model compression, we focus on pruning/sparsity. Pruning methods set a large fraction of model weights to zero and compactly represent the model in a sparse matrix format. These methods rank the model weights according



(a) Pruning at Initialization



(b) Iterative Pruning

Figure 2.5: Pruning at initialization prunes once before the start of model training while iterative pruning prunes repeatedly throughout the training procedure.

to a scoring function and prune a fraction of weights with the lowest scores. While there are numerous approaches to pruning and sparsification, we broadly divide current pruning literature along the following axes: structured vs. unstructured and pruning at initialization vs. iterative pruning. We describe these categories and their consequences. First, we compare structured and unstructured pruning -

- Structured pruning - In structured pruning, the scoring function ranks existing structures within the model architecture (filters, channels, layers, etc.). Structures are then pruned in their entirety by setting every parameter that exists in that structure to 0. Structured pruning is useful in accelerating training because block level sparsity can speed up matrix multiplication on GPUs. However, the downside to structured pruning is that there is less flexibility in pruning patterns, often resulting in a model that is significantly less accurate than its dense counterpart.
- Unstructured pruning - In unstructured pruning, the scoring function ranks individual weights within the model architecture. Weights with the lowest scores are set to 0 independently of other weights in the model. While unstructured pruning offers more flexibility in sparsity patterns and usually results in more accurate models, they offer few benefits in terms of training efficiency and inference latency. This is because random sparsity patterns require custom hardware to see noticeable speedups for operations like matrix multiplication. For general consumer hardware, there are few efficiency advantages to applying unstructured pruning.

The next dimension on which pruning techniques differ is pruning at initialization and iterative

pruning -

- Pruning at initialization - In pruning at initialization, model weights or structures are ranked according to the scoring function prior to any model training. Weights or structures with the lowest scores are immediately set to 0 and masked for the remainder of model training. This method allows you to pre-define a sparsity ratio and retain that sparsity ratio for the entire training procedure. However, the scores assigned to weights or structures at initialization may be inaccurate and fail to reflect their actual importances as a consequence of model training.
- Iterative pruning - In iterative pruning, at certain points during model training, the scoring function ranks the current weights or structures in the model. A smaller fraction of the model's weights are set to 0 and masked for the remainder of model training. In this way, the model becomes progressively more sparse throughout the training procedure. Iterative pruning tends to result in more accurate models because weights or structures are pruned based on their current value during model training and fewer weights are pruned at once. However, in initial rounds, the model is almost entirely dense resulting in little advantage over dense training until the model becomes sparser later in training.

We choose to focus on sparsity because recent literature has demonstrated that models can be significantly compressed using these techniques while retaining performance close to the original model. In contrast, techniques like quantization, which aim to represent weights in lower-bit precision formats, seem to degrade in accuracy more significantly with fewer memory savings. Even still, FLoSS must balance the utility-memory trade-offs described in the above definitions. Without a careful consideration of system design, it remains difficult to train a model using high levels of sparsity that is as accurate as its dense counterpart. We describe the procedure to achieve considerable performance and efficiency in our methods section.

Chapter 3

Related Works

3.1 Communication Efficiency in Federated Learning

A bottleneck in FL systems is the communication cost associated with training. In FL applications, clients communicate wirelessly with the central server. Additionally, in large-scale FL systems that utilize millions of clients, client devices can exist anywhere in the world. Taken together, this means that communication for FL training can be significantly slower than distributed data center training where devices are connected and local. Slow communication can make FL training impractical as long-lasting communication rounds could make it difficult for the model to converge in a timely fashion. For this reason, many FL applications consider communication efficiency an important principle when considering system design.

Modern approaches consider two different ways to tackle the issue of communication efficiency in FL, namely either reducing the number of communication rounds or reducing the size of communicated messages. Some approaches that fall into these categories are described below -

- Reducing the number of communication rounds - Approaches that aim to reduce the number of communication rounds focus on systematically improving model convergence time. For example, CMFL does not upload outlier updates from clients to keep updates in each round relevant to global model convergence. CA-FL and FL+HC use clustering to determine the most representative update from a set of clients and only sends that update to the central server. FedBoost uses an ensemble of models that converge faster than a single larger model to reduce the number of updates required for each of the ensemble models. Communication-Efficient Federated Learning uses a probabilistic model to select devices most likely to contribute to faster model convergence. Finally, methods like One-Shot FL and k-Fed propose performing all FL training using a single communication round, as opposed to iteratively updating the global model over multiple communication rounds. While some of these methods have empirically demonstrated improvements over vanilla FL, many have concrete failure modes that make it difficult to demonstrate their effectiveness in a broad array of real-world settings. Additionally, many of these methods require changes to the FL training procedure that are difficult to implement at scale.

- Reducing the size of communicated messages - Methods that reduce the size of communicated messages focus on sending or receiving partial updates or compressing the size of updates. LFL and FedPaq use quantization to represent model updates at a lower bit precision. FedMP, PruneFL, and model pruning for HFL all use various types of pruning during the communication phases of FL. Finally, FedKD and DS-FL use knowledge distillation to compress the model parameters and improve communication efficiency. We focus on these methods as they demonstrably reduce the communication cost at each communication round in training. However, we note that a significant drawback of these methods is that reducing the amount of information communicated at each round may degrade the model's utility. We design FLoSS in a way that balances communication efficiency and model performance across clients.

3.2 Fine-Tuning in Federated Learning

Training LLMs in FL is a difficult task due to computational constraints at the client-level and communication constraints across the network. These constraints necessitate the use of parameter-efficient fine-tuning (PEFT) methods like adapters, prompt-tuning, BitFit etc. By using these PEFT methods for local computation, resource-constrained clients can more effectively train models on local data. A few recent works have examine the idea of fine-tuning open source LLMs in a federated setting. FedPETuning uses methods like the Houlsby Adapter and prefix tuning to demonstrate efficiently fine-tuned LLMs in FL are robust to privacy attacks. Scaling Federated Learning proposes fine-tuning all the weights of model architectures like BERT and DistilBERT in FL using a fixed compute budget per client. FedPEFT benchmarks methods like bias-tuning and prompt-tuning in FL across a variety of dataset/model pairings. Each of these methods look to reduce the computational load of local client training. In our work, we consider dense PEFT as a naive baseline and study how to further reduce its message size using sparsity.

3.3 Pruning Adapters

Some recent literature has used pruning to enhance the efficiency of adapters. Specifically, SparseAdapter proposes pruning adapters once at initialization. The pruned weights are set to 0 for the entire training procedure. Thus, sparsity is only applied once prior to the beginning of training and the sparsity pattern never changes throughout the training process. AdapterLTH (Lottery Ticket Hypothesis) performs iterative pruning by alternating between pruning away a small fraction of the lowest scoring weights and retraining the remaining weights of an adapter module such as LoRA. While these works improve the storage efficiency of LoRA, they otherwise have marginal practical benefits in centralized settings. First, LoRA parameters can be merged back into the model backbone to eliminate storage costs after training. Second, Unstructured sparsity often requires specialized hardware and software to accelerate computation. Otherwise, training and inference are no more efficient than that of a dense counterpart. We argue that combining unstructured sparsity with LoRA is particularly effective for handling issues of communication in FL.

Chapter 4

Methods

This section details the combination of adapter methods and sparsity techniques we employ in FLoSS. We additionally highlight the importance of each step in resolving real-world bottlenecks, such as on-device computation and communication constraints, for FL training. Finally, we describe a set of benchmarks we compare against to demonstrate the effectiveness of our training procedure.

4.1 Federated LoRA

FLoSS utilizes the LoRA adapter in a federated setting. Federated LoRA training functions as follows. The training procedure starts with a pretrained LLM that exists at the central server. Each client then downloads a copy of the weights of the pretrained LLM. While this an expensive operation, we note that in this training procedure the full weights of the LLM are only ever communicated once. Additionally, since the full LLM weights are only ever communicated during a download phase, and download is typically much faster than upload, we can afford to incur this one-time communication cost during this training procedure. The central server then initializes LoRA parameters in each transformer block of the pretrained LLM. FL training then proceeds similarly to the description provided in section 2.1 with one critical difference - only the LoRA parameters are communicated between the central server and the client devices. At each communication round, the sampled clients only download a copy of the global model's LoRA parameters. If the sampled clients do not have initialized LoRA parameters for their local model, they initialize fresh LoRA parameters. Then all sampled clients replace the weights of their LoRA parameters with the weights of the downloaded LoRA parameters. The sampled clients then train only the weights in the LoRA parameters, freezing the remaining LLM weights as described in section 2.2. Once local training is complete, the sampled clients send only the LoRA parameters back to the central server where they are aggregated into the new global model.

Using LoRA for FL is particularly useful as it enables efficient on-device training. Training the full LLM at the client level would be difficult given memory and computation constraints. In cross-device FL settings, client devices tend to be especially resource-constrained. As such, reducing the computational load of FL training is important in this setting.

4.2 Sparsity for LoRA

While LoRA enables on-device training by significantly reducing VRAM usage, LoRA parameters can still be expensive to communicate. LoRA adapters are inserted for each weight matrix in the MSA layer and for each transformer block in the model. Communicating all these parameters is difficult on a wireless network where communication, and specifically the upload phase, is slow and time-consuming. Thus, it is important to retain the on-device computational benefits of LoRA training while reducing communication latency for this training procedure. We reduce communication cost by applying top-k sparsity to LoRA during the download and upload phase of FL training.

To perform top-k sparsification consider a weight matrix $W \in \mathbb{R}^{d\times k}$ with n nonzero entries. We define a sparsity ratio $\alpha \in [0, 1]$. We apply the top-k function to $|W|$ where $k = \alpha \cdot n$ and retrieve the indices of these top-k values. We call this set of indices $I = \{(y_i, x_i) \mid y_i \in [0, d-1],\ x_i \in [0, k-1]\}$. We then define a binary mask $M \in \mathbb{R}^{d\times k}$ where $M[y_i, x_i] = 1$ if $(y_i, x_i) \in I$ and $M[y_i, x_i] = 1$ otherwise. W is then updated as $W \odot M$. This functions to mask the smallest weights in W.

In our training scheme, top-k sparsity is applied to the LoRA adapters prior to download and prior to upload. In this way, fewer weights are communicated in each phase, reducing communication latency as a result. There are a few important design considerations in determining how sparsity should be specifically applied to LoRA. First, the sparsity ratio can be different for download and upload. This is because download bandwidth is typically much greater than upload bandwidth (the average download bandwidth for a wireless network is 200 Mbps while the average upload bandwidth for a wireless network is 20 Mbps). Therefore, being able to configure separate sparsity ratios is important as sparser downloads may be less helpful than sparser uploads. Second, unstructured sparsity is more useful in our setup than structured sparsity. An important distinction is that FLoSS does not retain sparsity during client local training. This is because sparsity would be unhelpful in accelerating training on devices such as smartphones or IoT devices where the hardware is not specifically designed to handle sparse computation. Therefore, dense training would yield results just as quickly and much more accurately than sparse training at the client level. Since unstructured sparsity has been shown to boost performance relative to structured sparsity, and training acceleration is not a consideration, unstructured sparsity is the preferred sparsity method for communication in FLoSS.

4.3 FLoSS Algorithm

FLoSS combines federated LoRA with sparsity to perform communication-efficient LLM training. At each communication round, the central server will sparsify the LoRA parameters of the global model according to a download sparsity ratio d_{down}. Sampled clients then download these sparse adapters and train them locally without any sparsity applied. Before uploading the locally trained adapters, clients sparsify the adapters according to an upload sparsity ratio d_{up}. The communication round ends with the central server aggregating these sparse adapters.

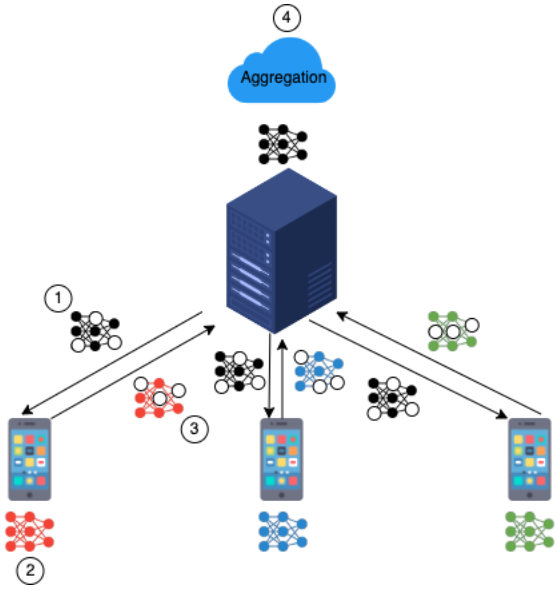


Figure 4.1: FLoSS procedure involves (1) sparsifying adapters prior to download (2) training dense adapters (3) sparsifying adapters prior to upload (4) aggregating using FedAdam.

FLoSS offers a few key advantages for federated LLM training. First, FLoSS enables more efficient on-device training for LLMs. It does so by training only LoRA adapter weights and limiting local training to a single epoch in each communication round for all clients. Since cross-device settings usually involve many clients where each client has limited data, we argue that this training scheme is feasible at the client-level. Second, FLoSS reduces communication latency despite the large number of adapter weights being communicated in each round. By sparsifying adapters prior to communication, fewer weights are communicated, and communication can occur more quickly. Finally, FLoSS retains model performance by allowing dense fine-tuning at the client-level. We demonstrate the importance of this last feature in an ablation study that identifies the consequence of various pruning methods on model performance. Ultimately, FLoSS achieves high communication efficiency without sacrificing model performance.

4.4 Benchmarks

We present experiments on three datasets: CIFAR10, 20NewsGroups, and Reddit. We resize the CIFAR10 images to 224×224 to match ImageNet, the pretraining dataset for the ViT model architecture we chose. We use the GPT2 tokenizer to preprocess the examples of 20Newsgroups and Reddit into sequences with length 128 and 25 respectively. We partition CIFAR10 and 20NewsGroups across the client devices. As described in the Results section, we test both I.I.D. clients as well as non-I.I.D. clients for partitioning both datasets. The Reddit comments are naturally partitioned by user.

In all experiments, we sample 5 clients at each round and perform one epoch of local train-

ing with a batch size of 16. We fine-tune all models for 200 rounds. For the pretrained models, we used ViT-B-16 (85M params) and GPT2-Small (124M params). For all datasets, we report the accuracy on the validation partition. More details on the task setups can be found in Table 4.1.

Dataset	Backbone	Task	#Clients	#Examples	#Classes
CIFAR10	ViT-B-16	Image Classification	500	50K	10
20NewsGroups	GPT2-Small	Sequence Classification	350	20K	20
Reddit	GPT2-Small	Next Token Prediction	40K	1.1M	50257

Table 4.1: Statistics of the datasets used in the experiments.

We compare FLoSS against two other sparse LoRA methods, SparseAdapter and AdapterLTH. Both these methods are described in the Related Works section. To use AdapterLTH in FL, we consider training LoRA weights A and B using FedAdam. After each aggregation round, we apply increasingly sparse masks to the LoRA weights. We use the efficient "fine-tuning" version of LTH which continues training from the pruned state rather than rewinding the weights after pruning. This allows the model to recover from pruning within fewer rounds and is necessary to keep communication costs competitive with the dense LoRA baseline. For both of these methods our choice of scoring function is top-k applied to the magnitude of the weight. This allows for a direct comparison to FLoSS which uses the same unstructured scoring function but only sparsifies communication as opposed to sparsifying both communication and model training. For these pruning methods, we perform an initial round of dense LoRA training, so the B adapter is non-zero and can be effectively pruned using our score function.

Algorithm 1: PyTorch-like LoRA training with FedAdam and sparse communication

Require: $d_{\text{down}}, d_{\text{up}}$ (download and upload sparsity ratio)
$P \leftarrow$ Initialize LoRA parameters
`optim` $\leftarrow$ `torch.nn.optim.Adam(params=`P`)`
for $r = 1, ..., R$ **do**
  $M_{\text{down}} \leftarrow$ mask of top d_{down} fraction entries of P by magnitude
  Sample clients $c_1, ..., c_n$ uniformly at random without replacement
  for $i = 1, ..., n$ *in parallel* **do**
    $P_i = P \odot M_{\text{down}}$ # sparse download
    $P_i' \leftarrow$ update P_i with 1 SGD epoch on data of c_i # fine-tuning all entries of P_i
    $\Delta P_i \leftarrow P_i - P_i'$
    $M_{\text{up},i} \leftarrow$ mask of top d_{up} fraction entries of ΔP_i by magnitude
    $\Delta P_i \leftarrow \Delta P_i \odot M_{\text{up},i}$ # sparse upload
  `optim.grad` $\leftarrow \frac{1}{n}\sum_{i=1}^{n} \Delta P_i$ # set Adam pseudo-gradient
  `optim.step()` (updates P) # take one step of Adam

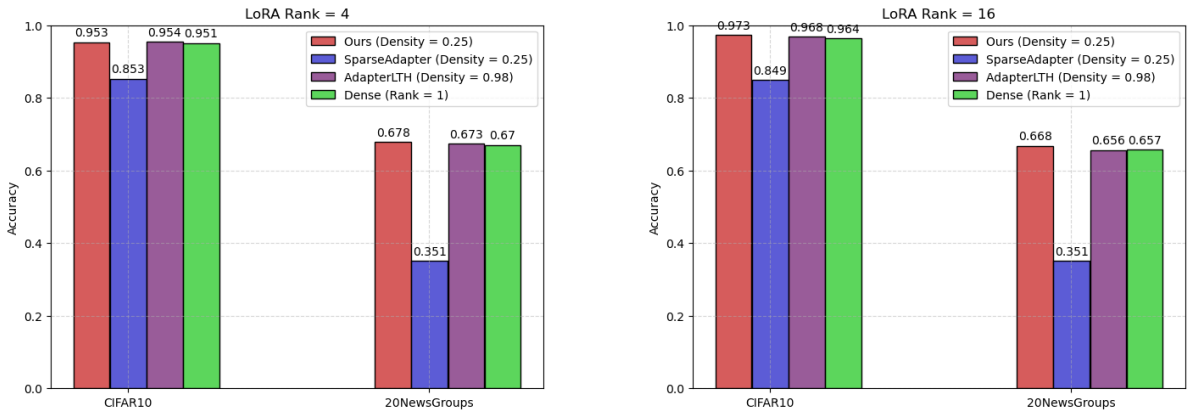
Chapter 5

Results

We describe our key findings when running FL training experiments with FLoSS. There are two key metrics by which we can measure the effectiveness of this training procedure: model performance and communication latency. The first section compares the performance of models trained with FLoSS with both dense training as well as other pruning benchmarks. We highlight that FLoSS achieves performance comparable to dense training, even at high levels of sparsity, while significantly outperforming other pruning benchmarks. The second section serves to demonstrate the communication benefits of FLoSS by reducing communication time in realistic wireless networks. Finally, we provide heuristics for automatically determining a rank and download/upload sparsity configuration for achieving accurate training given a certain communication budget.

5.1 Model Performance with Fixed Communication

We compare the performance of models trained with FLoSS to dense LoRA training and other pruning methods in a federated setting across our various configurations of datasets and model architectures. While dense adapter training will necessarily communicate the entire adapter, in order to compare model performance across communication-efficient pruning methods we fix a single rank and download/upload sparsity configuration across methods. For FLoSS we fix both the download and upload sparsity to be 0.25 and SparseAdapter prunes to 0.25 sparsity at initialization. AdapterLTH is harder to fix in this same way as it iteratively prunes throughout training. As we train for 200 communication rounds, we set AdapterLTH to use incremental sparsity ratio 0.98. This way, on average across 200 communication rounds, AdapterLTH is communicating approximately 0.25 of the weights. Comparing in this way allows us to test model performance across the various training methods with fixed communication. For this reason, we also include results for dense LoRA with rank that is $\frac{1}{4}$ of the original rank as this is equivalent to communicating 0.25 of the weights with our original LoRA rank. While these numbers appear arbitrary, we note that the method is robust to multiple settings of these hyperparameters. We include additional experiments with different rank and download/upload sparsity ratio configurations can be found in the Appendix.



(a) LoRA Rank = 4

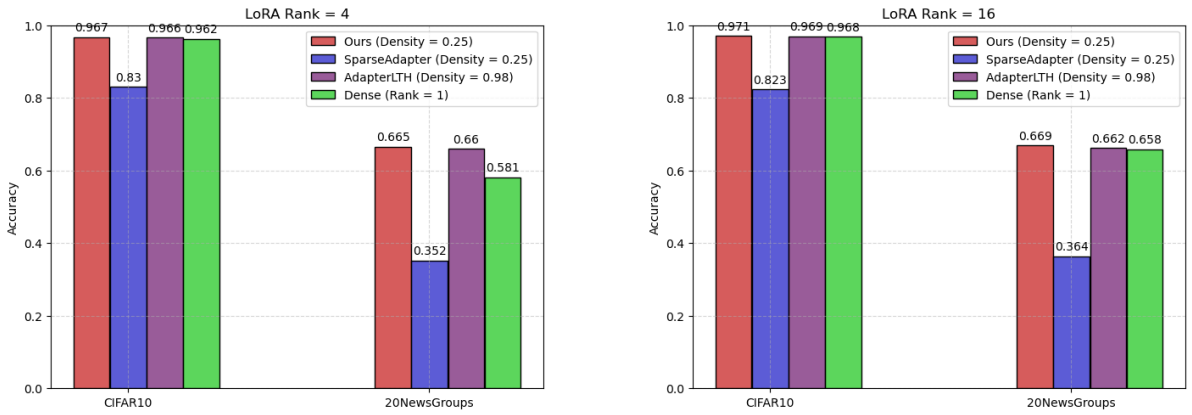
(b) LoRA Rank = 16

Figure 5.1: Comparison of communication-efficient LoRA methods in FL.

The results in Figure 5.1 demonstrate that FLoSS outperforms other efficient fine-tuning methods in a federated setting. Specifically, FLoSS performs significantly better than the one-shot pruning method specified in SparseAdapter and marginally better than AdapterLTH and dense training of a LoRA adapter with $\frac{1}{4}$ rank. These findings are consistent in experiments in CIFAR10, 20NewsGroups, and Reddit demonstrating that the method of sparsifying an adapter prior to download/upload is an effective way to reduce communication costs for multiple tasks. Additionally, this sparsification works even with smaller ranks (e.g. rank=4) meaning that we can get especially efficient communication by using a combination of LoRA where the rank is significantly smaller than the embedding dimension in conjunction with sparse updates. While we note that some of these performance benefits appear to be marginal at first, we show in later sections that FLoSS offers significant communication benefits in comparison to the other methods described as the performance of the method does not degrade with extreme levels of sparsity and different download/upload sparsity ratios.

5.2 Robustness to Statistical Heterogeneity

While Reddit naturally has non-I.I.D clients as a consequence of being partitioned by user, CIFAR10 and 20NewsGroups do not have meta-data regarding the user that produced certain images. This means that if the datasets are partitioned across clients in a random fashion, the clients will generally have I.I.D. data. This may not be representative of real-world FL environments where clients may have non-I.I.D. data and training can be affected by statistical heterogeneity. To incorporate statistical heterogeneity for CIFAR10 and 20NewsGroups we partition the datasets across clients based on the ground-truth label for each image. We follow the procedure described by Hsu et al. using a Dirichlet($\alpha = 0.1$) to generate synthetic non-I.I.D. client data from CIFAR10 and 20NewsGroups. In this way, approximately 90% of the examples for each client are of a single label. We repeat the experiments in the last section for CIFAR10 and 20NewsGroups with non-I.I.D. clients.



(a) LoRA Rank = 4

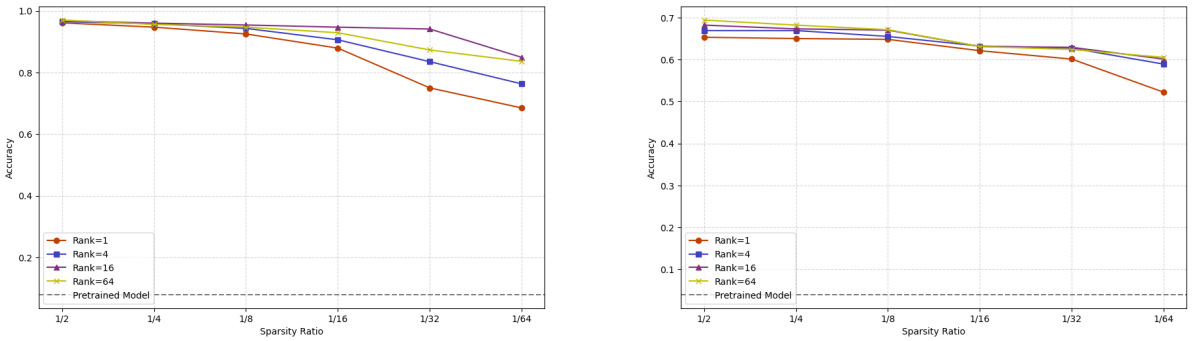
(b) LoRA Rank = 16

Figure 5.2: Comparison of communication-efficient LoRA methods in FL with non-I.I.D clients.

Based on the results in Figure 5.2, FLoSS performs well even in the presence of extreme statistical heterogeneity. Despite clients primarily sampling examples from a single label, there is little degradation in the model's utility in comparison to the I.I.D. setting. Additionally, FLoSS still outperforms the other efficient fine-tuning methods demonstrating that the method still results in an expressive, effective model despite communicating very few parameters between the central server and client devices in any communication round. Our final observation is that Hsu et al. reports that statistical heterogeneity has an adverse impact on a CNN architecture trained on CIFAR10 for FL. In comparison, we find that fine-tuning a pretrained LLM on a similarly heterogeneous CIFAR10 partitioning for FL is not hindered by the same performance losses. This suggests that LLMs may offer a potential solution to the performance problems that arise as a result of statistical heterogeneity in FL.

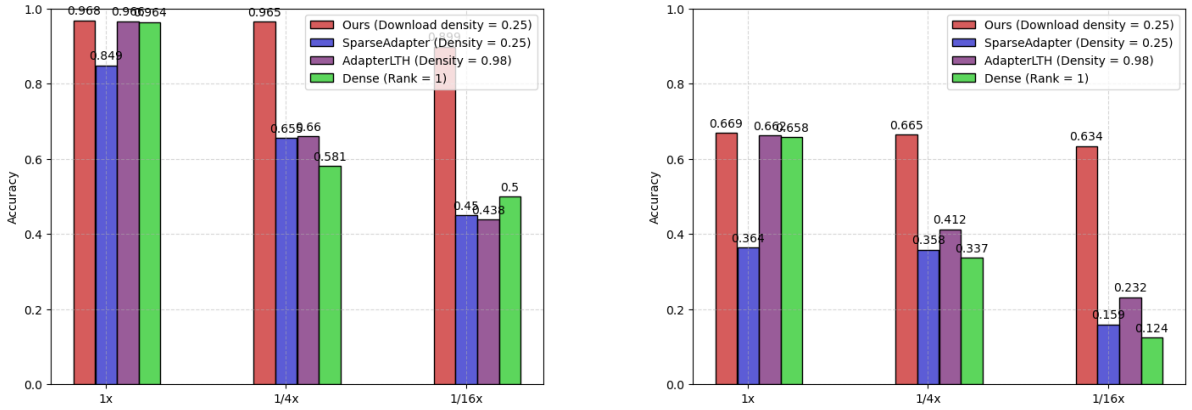
5.3 Improvements to Communication Efficiency

While FLoSS marginally outperforms other efficient training methods on evaluation data, FLoSS improves the communication-efficiency of training significantly in comparison to other methods. There are two critical reasons why FLoSS is able to reduce communication costs considerably: robustness to extreme sparsity and variable download/upload sparsity. First, FLoSS is able to utilize extremely sparse communication (e.g. approximately 0.01 sparsity ratio) and retain performance. In comparison, SparseAdapter noticeably degrades in performance even at sparsity ratios of 0.25 as demonstrated in the previous sections. AdapterLTH relies on pruning very few additional parameters at every communication round. However, this iterative pruning technique is not well-suited for extreme sparsity. In order for AdapterLTH to produce an average sparsity ratio of 0.01 across 200 communication rounds, the method would have to iteratively prune 50% of the weights at every communication round. In the last communication round, only 1.24e-60 of the LoRA weights would remain, leading to a model that is no more accurate than the pretrained model. Figure 5.3 demonstrates that across multiple ranks, FLoSS can support extreme sparsity and result in a model significantly better than the pretrained model.



(a) CIFAR10 (b) 20NewsGroups

Figure 5.3: Model performance with different LoRA ranks at various sparsity levels.



(a) CIFAR10 (b) 20NewsGroups

Figure 5.4: Model performance with separate download/upload sparsity ratios.

The second reason why FLoSS is able to significantly cut down on communication costs is because the method is able to define separate download and upload sparsity ratios. In wireless networks, upload is significantly more expensive than download (with upload sometimes being up to 10x slower). In cases where upload speed is significantly reduce in comparison to download speed, methods like AdapterLTH and SparseAdapter are too rigid in that AdapterLTH can only prune prior to download and SparseAdapter can only prune prior to model training. Figure 5.4 considers cases where upload is as slow as download, 4x times slower than download, and 16 times slower than download. In each case, we set the upload sparsity ratio of floss to be the download sparsity ratio, $\frac{1}{4}$ of the download sparsity ratio, and $\frac{1}{16}$ of the download sparsity ratio respectively. We then train each method with a communication budget that is equal to 50 rounds of dense LoRA training. We find that when upload is slower than download, the other efficient training methods degrade in performance significantly whereas FLoSS retains high performance with its ability to set adaptive sparsity ratios. This finding suggests that FLoSS is better suited for real-world FL configurations that have disparate download and upload speeds.

Chapter 6

Conclusion

This thesis examines the problem of training LLMs in a federated environment. The scale of LLMs presents a problem in FL settings due to the limited compute on client devices and limited communication across the network. We present a few critical steps that can be taken to tackle these bottlenecks and enable efficient training. First, we employ adapter methods, specifically LoRA, to reduce the computational load of local model training. LoRA is especially useful in FL as client devices do not have train the full model parameters of the LLM and the adapters can be merged back into the backbone of the model to eliminate and storage and inference costs after training. We augment the communication efficiency of LoRA using sparsity applied only to communication. Crucially, we find that applying sparsity in this way is critical to retain performance as methods that prune adapters at initialization or iteratively prune adapters throughout training do not perform as well with limited communication budgets or cases where upload sparsity is slower than download sparsity. These limitations suggest that sparsity must be carefully applied during the training process to ensure that communication-efficiency does not degrade model utility. A few crucial directions remain to be explored in the context of efficient LLM training for FL. Namely, while LoRA certainly makes local computation easier for client devices, methods to further enable LLM training in resource-constrained environments may improve the functionality of this method. For example, quantization or mixed precision training may enable even faster local compute. Next, methods to automatically configure the rank and sparsity ratios for FLoSS are important in preventing expensive FL hyperparameter tuning. Our experiments suggest that certain rank and sparsity configurations do not perform as well as others within the same communication budget. Therefore, being able to automatically determine the optimal value for these hyperparameters prior to training may be crucial in extracting the full value of LLMs in this setting. n the future, we aim to investigate such questions and design methods to make high-quality models more accessible to low-resource users.

Bibliography