FedBiOT: LLM Local Fine-tuning in Federated Learning without Full Model

Feijie Wu*
Purdue University
West Lafayette, IN, USA
wu1977@purdue.edu

Zitao Li
Alibaba Group
Bellevue, WA, USA
zitao.l@alibaba-inc.com

Yaliang Li
Alibaba Group
Bellevue, WA, USA
yaliang.li@alibaba-inc.com

Bolin Ding
Alibaba Group
Bellevue, WA, USA
bolin.ding@alibaba-inc.com

Jing Gao
Purdue University
West Lafayette, IN, USA
jinggao@purdue.edu

Abstract

Large language models (LLMs) show amazing performance on many domain-specific tasks after fine-tuning with some appropriate data. However, many domain-specific data are privately distributed across multiple owners. Thus, this dilemma raises the interest in how to perform LLM fine-tuning in federated learning (FL). However, confronted with limited computation and communication capacities, FL clients struggle to fine-tune an LLM effectively. To this end, we introduce FedBiOT, a resource-efficient LLM fine-tuning approach to FL. Specifically, our method involves the server generating a compressed LLM and aligning its performance with the full model. Subsequently, the clients fine-tune a lightweight yet important part of the compressed model, referred to as an adapter. Notice that as the server has no access to the private data owned by the clients, the data used for alignment by the server has a different distribution from the one used for fine-tuning by clients. We formulate the problem into a bi-level optimization problem to minimize the negative effect of data discrepancy and derive the updating rules for the server and clients. We conduct extensive experiments on LLaMA-2, empirically showing that the adapter has exceptional performance when reintegrated into the global LLM. The results also indicate that the proposed FedBiOT significantly reduces resource consumption compared to existing benchmarks, all while achieving comparable performance levels.

CCS Concepts

• **Computing methodologies** → **Distributed algorithms**; *Natural language generation*; • **Information systems** → *Language models.*

Keywords

Federated Learning; Large Language Models

*Work was done while Feijie Wu was an intern at Alibaba Group.

KDD '24, August 25–29, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-0490-1/24/08
https://doi.org/10.1145/3637528.3671897

ACM Reference Format:
Feijie Wu, Zitao Li, Yaliang Li, Bolin Ding, and Jing Gao. 2024. FedBiOT: LLM Local Fine-tuning in Federated Learning without Full Model. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3637528.3671897

1 Introduction

The recent advancements in large language models (LLMs) have demonstrated incredible performance in various tasks, such as question-answering and problem-solving. This success owes to the pretraining on large datasets, covering a wide range of linguistic patterns and general knowledge. However, in specific domains such as legal advice [7, 23] and medical diagnosis [27, 32, 39], LLMs may not provide professional responses because the terminology and context significantly differ from general language use. To address this limitation and enable the generation of domain-specific content, it becomes imperative to fine-tune LLMs with relevant data. This fine-tuning process allows the models to learn from the specific instances and nuances of the target application, ensuring their capability within specialized fields. The quality and quantity of the task-specific data are directly related to the performance of the fine-tuned model on downstream tasks: large and well-labeled data can significantly improve the model, while small and irrelevant data can only benefit the model marginally. However, there are many cases where task-specific data are possessed by multiple data parties, while each of them may have a limited number of samples that can be used to fine-tune LLMs. For example, a hospital in a rural area may only have a limited number of lung cancer cases recorded in its own system; if an LLM is only fine-tuned on one set of those cases, it may not obtain comprehensive knowledge and easily be overfitted.

To incorporate all the distributed data in the fine-tuning of LLMs, one may consider the batch fine-tuning as follows. If we demand all the data owners to share their data with the LLM server, then LLM fine-tuning could be conducted at the server side. For example, some LLM owners offer fine-tuning APIs as services, but the users must pack their data as files and upload them to use a black-box fine-tuning [24]. Apparently, this setup is not applicable to users who have privacy concerns. Especially, some businesses are subject to data privacy regulations [2, 8], which makes it challenging to share local data with LLM server.

Therefore, a more practical setting is to let individual data owners keep their data locally, run fine-tuning locally and aggregate the fine-tuning results at the LLM server. This fits well the *federated learning* (FL) framework, which is a distributed paradigm that places a paramount emphasis on preserving privacy. Its conventional algorithms, such as FedAvg [22], are considered practical solutions to overcome data barriers across different data owners. In this paradigm, data owners are treated as clients, and an LLM server coordinates the computation. The standard FL workflow involves three steps repeatedly: (i) The server distributes the global model to all clients; (ii) Each client trains the model locally for multiple iterations and sends the updated model to the server; (iii) The server aggregates the models from the clients and updates the global model accordingly. Despite the potential of this method to facilitate collaborative fine-tuning of an LLM without sharing local data, its feasibility is hindered by two main limitations:

- **Access to full model of state-of-the-art LLMs:** There exist some open-source LLMs whose model the public can download and have full access to their parameters. However, the most recent and powerful versions of LLMs are usually closed-sourced, i.e, the architecture and parameters are not available to the public. The best closed-source LLMs still have leading performance on a wide range of language tasks, and its leading edge can be maintained or even enhanced after fine-tuning, making it a better choice. As aforementioned, using the blackbox fine-tuning service provided by these closed-source LLMs often violates data users' privacy requirements. Therefore, a federated learning framework that conducts collaborative fine-tuning with the assumption of no access to the full model of LLMs at the client side is more desirable.
- **Computation and communication costs:** Existing federated learning framework could also suffer from the computation and communication challenges when conducting collaborative fine-tuning on LLMs. The fine-tuning process for LLMs entails substantial computational demands and communication costs due to the vast number of trainable model parameters. Clients with limited computational power may struggle to perform complex model updates, leading to prolonged training times or potential disruptions. The transfer of expensive models between the server and the clients also incurs substantial communication costs, leading to substantial bandwidth consumption and increased communication latency. At the server side, there could be network congestion when clients send back their updated huge amount of parameters concurrently.

In this paper, we aim to tackle these two challenges and propose to design an effective and practical collaborative LLM fine-tuning framework. To address the first challenge, We follow the setting proposed in offsite-tuning [43] and its federated version FedOT [13]. We assume that the LLM owner does not collect data directly from clients but serves as the server in FL, who can use a public dataset to distill her LLM and aggregate some local updates on part of the model from clients; multiple clients want to collaborate on fine-tuning for similar downstream tasks. Different from the classic FL setting [18, 35, 47], we do not assume the data distribution on clients or the public data owned by the server to be the same. Our goal, in general, is to provide a framework for collaborative clients to fine-tune without access to full LLM or sharing local data directly. More importantly, the fine-tuned model can still achieve better performance than fine-tuning LLM locally with their local data exclusively.

Although FedOT [13] was developed for this objective, it could incur significant computational and communication costs, thereby suffering from the second challenge. In light of this challenge, we propose to integrate various parameter-efficient fine-tuning (PEFT) techniques into the proposed FL framework. Specifically, the server employs linear dropout to compress the LLM, integrates LoRA [11] to reduce the trainable parameters, and divides it into two components: an emulator and an adapter. The emulator retains a consistent representation of the raw model on the server's dataset, while the adapter assimilates domain-specific linguistic patterns from the clients' local datasets. Considering the significant distribution shift between the clients' datasets and the server's dataset, we separate the fine-tuning of these two components into two processes during FL training, i.e., the clients perform multiple local updates to fine-tune the adapter, and the server distill the emulator from the original LLM while aggregating the updated adapters from the clients. To this end, a bi-level optimization is formulated.

This design, named **Fed**erated **Bi**-level **O**ffsite **T**uning (FedBiOT), offers twofold advantages from the clients' perspectives. Firstly, instead of loading the complete model, clients load a compressed version with fewer layers, considerably reducing computation costs. Secondly, clients exclusively fine-tune the adapter, affecting only the last few layers of the LLM and thereby minimizing computation and communication expenses.

Contributions. Throughout the paper, our contributions are highlighted as follows:

- We propose an algorithm FedBiOT that avoids full model fine-tuning and significantly reduces the communication and computation overhead. To the best of our knowledge, this is the first work that addresses both the aforementioned two challenges in the federated LLM fine-tuning framework. With our proposed framework, clients' data are ensured to be kept locally and computation and communication burden is significantly reduced.
- We formulate a bi-level optimization problem that enables the LLM fine-tuning without access to the full model. By partitioning the compressed model into the adapter and the emulator, the emulator acts as a simulator of the original raw model, while the adapter adeptly learns domain-specific linguistic patterns with clients' local datasets. To this end, we realize that fine-tuning the compressed model is equivalent to the refinement of the counterpart of the complete LLM.
- We conduct extensive experiments on LLaMA-2 for fine-tuning with three tasks, i.e., code generating, math problem solving, and question answering. The empirical studies also demonstrate that the proposed approach has significant improvement over all these tasks compared with the baseline approaches in terms of computation and communication overheads and final accuracy.

2 Preliminary

2.1 Traditional FL Formulation

Consider there is an FL system with a total of M clients, denoted by $[M]$. Each client $m \in [M]$ holds a local dataset $\mathcal{D}_m$. A client's

local loss is defined as

$$F_m(\boldsymbol{w}) := \frac{1}{|\mathcal{D}_m|} \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{D}_m} f\left(\mathcal{M}(\boldsymbol{x};\boldsymbol{w});\boldsymbol{y}\right), \tag{1}$$

where $\mathcal{M}(\boldsymbol{x};\boldsymbol{w})$ is the output on a given model parameterized by $\boldsymbol{w}$ and an input $\boldsymbol{x}$. The loss function f is defined on the model output and the ground truth $\boldsymbol{y}$. In this dataset, we assume that the ground truth $\boldsymbol{y}$ is part of the input $\boldsymbol{x}$, where a sequence of tokens in the input is used to predict the next token, and the ground truth is used to identify the part needing to be predicted by the model. Such a dataset is commonly adopted in previous works to fine-tune an LLM [25, 40]. Then, based on the definition, a conventional FL system aims to find an optimal model across all clients, which is formulated as

$$\min_{\boldsymbol{w}\in\mathbb{R}^d} F(\boldsymbol{w}) = \sum_{m\in[M]} p_m F_m(\boldsymbol{w}), \tag{2}$$

where $p_m = |\mathcal{D}_m|/|\mathcal{D}|$ for all $m \in [M]$, where $\mathcal{D}$ represents the entire training dataset, i.e., $\mathcal{D} = \cup_{m\in[M]}\mathcal{D}_m$. Generally, this problem can be optimized by different FL algorithms [12, 15, 38] repeating the following paradigm until convergence:

- **Step 1:** At the beginning of each round t, the server broadcasts the trainable model parameters $\boldsymbol{w}^{(t)}$;
- **Step 2:** After receiving the model $\boldsymbol{w}^{(t)}$, each client $m \in [M]$ performs multi-step local updates on $\boldsymbol{w}^{(t)}$ to obtain $\boldsymbol{w}_m^{(t)}$;
- **Step 3:** The server collects the locally updated model parameters $\boldsymbol{w}_m^{(t)}$ from clients and aggregates them into a single global model $\boldsymbol{w}^{(t+1)}$ for next round.

Applying PEFT to federated LLM fine-tuning. The existing FL algorithms [1, 9, 36, 37, 42] are confronted with computation and communication bottlenecks when fine-tuning an LLM. To mitigate the limitations, researchers have extended existing parameter-efficient fine-tuning (PEFT) approaches to FL, named FedPEFT [30, 45, 49]. These methods minimize the number of trainable parameters by introducing a PEFT module and keeping the original LLM parameters unchanged. By focusing local updates exclusively on the PEFT module rather than the entire model, these methods effectively reduce computational load and support larger batch sizes on a single GPU. Additionally, the FL server merely aggregates the updated parameters of a given model, thus obviating the need to transmit unchanged parameters and minimizing communication overheads.

Nevertheless, FedPEFT is still confronted with the intrinsic challenge wherein clients face obstacles in loading an LLM due to its substantial computation prerequisites. For instance, the loading of a full-precision LLaMA-2-7B necessitates a memory capacity of no less than 28 GB.

2.2 Related Work

The era of LLM poses the necessity of model privacy protection, where the details of LLM cannot be visible to the clients. To this end, Xiao et al. [43] proposes a method named Offsite-tuning under the scenario where there is a server (a.k.a. LLM owner) and a client, while Kuang et al. [13] extends this work to an FL version and names it as FedOT. They achieve model privacy protection by compressing the model, where only some layers are visible to the clients. However, these works require the preservation of a large number of layers to guarantee the performance, hindering the effectiveness of model privacy protection. In contrast, our work only discloses a few model parameters of the original LLM to the clients, i.e., the clients only know the adapter parameters that come from the original LLM, while the emulator parameters have been updated and different from the original LLM. Besides, neither offsite-tuning [43] nor FedOT [13] consider the difference between alignment data on the server and the fine-tuning data on clients. In contrast, the bi-level optimization problem proposed in our work naturally considers this factor and we design updating rules based on it.

Black-box is also a practical way to protect model privacy, where the clients access the LLM via an API, and they cannot fine-tune the LLM. Therefore, the optimization solely relies on prompt-based learning [14, 16, 20, 28]. In the context of FL, there are two typical works, namely, Fed-BBPT [19] and FedBPT [29]. These two works guarantee the model privacy in FL, but they should transmit the prompt together with the input to the LLM owner, leading to concerns about data privacy when the input contains sensitive information, violating the requirement of FL. In contrast, the proposed FedBiOT will not lead to this concern because its training is fully on the clients such that the data are never shared with others.

3 FedBiOT

Given that some clients may be unable to load a complete LLM, this section introduces an algorithm designed to enable these clients to fine-tune the LLM without requiring access to its full version. In other words, our goal is to refine the part of a compressed model that should yield performance comparable to fine-tuning its counterpart within a full model. To accomplish this, the server initially compresses the LLM and divides it into two distinct components, each serving specific functions. The first component, termed an emulator, is tasked with replicating the behavior of the uncompressed LLM. The second component, referred to as an adapter, focuses on adeptly acquiring domain-specific linguistic patterns from clients. Upon reintegrating the adapter into the uncompressed emulator, its performance should demonstrate significant improvement compared to the original LLM.

However, direct fine-tuning of the adapter on its models presents two significant limitations. Firstly, given that a single layer of a large language model (LLM) comprises millions of parameters, such as the decoder layer of LLaMA-2 with 202 million parameters, the adapter's parameter count is immense. This necessitates clients to possess powerful computational equipment to handle the fine-tuning of the layer. Additionally, transmitting the layer updates to the server poses another bottleneck, particularly in scenarios with unreliable network connections or limited bandwidth, hindering the smooth transmission of updates to the server.

To address these constraints, we integrate LoRA [11], a PEFT module, into our proposed method. LoRA significantly reduces the number of tunable parameters, with a LoRA module for LLaMA-2 comprising 0.13 million trainable parameters, which is merely 0.06% of the original layer's size. Consequently, the communication cost experiences a remarkable reduction of 99.94% compared to transmitting a full layer.

Organization. In the subsequent sections, we will delve into the concrete details of the algorithm design. Specifically, Section 3.1 illustrates how the compressed model is prepared. Following that, Section 3.2 discusses the problem formulation for the aforementioned objectives. On top of this, Section 3.3 and Section 3.4 outline the detailed steps of the proposed algorithm, namely local updates and server aggregation, showcasing the seamless integration of LoRA modules. Full implementation of the pseudocode is given in Algorithm 2.

3.1 Compressed Model Preparation: Linear Dropout

Suppose a pre-trained LLM has a total of n layers of transformers. In the work, a repeatedly used operation is layer extraction, which extracts some layers out of the total n layers of transformers to form a submodel. We denoted this by a function $\mathsf{LayerExtract}(\mathcal{M}, L)$, which means extracting the layers with indices in $L \subseteq [n]$ from the model $\mathcal{M}$. The function consists of the following three steps, and its pseudocode implementation of the first two steps is presented in Algorithm 1.

Step 1: Identify the adapters in the original model. We choose the bottom few layers[1] of the original LLM as the adapter. To be more specific, suppose the size of the adapter is a, and denote the adapter as $\mathcal{A}$. Therefore, $\mathcal{A} \leftarrow \mathsf{LayerExtract}(\mathcal{M}, L_{\mathcal{A}})$ with the layer indices $L_{\mathcal{A}} = \{n-a+1, \ldots, n\}$. We denote $\boldsymbol{w}_{\mathcal{A}}$ as the parameters of $\mathcal{A}$. The $\boldsymbol{w}_{\mathcal{A}}$ are also the trainable parameters that can be fine-tuned by clients. The remaining part of the model is demoted as $\mathcal{E}^* = \mathcal{M} \setminus \mathcal{A}$.

The choice of adapters brings two advantages. First, regarding the computation constraints of the clients, this proposed adapter is computation-efficient because it only needs to store the activations of transformers in the last few layers, leading to a lower memory cost. Second, as the adapter focuses more on domain-specific features, it is eco-friendly to spend the effort fine-tuning the last few layers. The conclusion is drawn from a well-known finding [46] in neural networks that the first few layers tend to learn general features while the last layers encode specific ones.

Step 2: Layer dropout to form emulator. Inspired by the experimental results presented by Xiao et al. [43], we form an emulator by means of a uniform layer dropout [26] from the remaining part $\mathcal{E}^*$. Therefore, the emulator is a sub-model obtained as $\mathcal{E} \leftarrow \mathsf{LayerExtract}(\mathcal{E}^*, L_{\mathcal{E}})$. Denote there are $n_{\mathcal{E}^*}$ layers transformer in $\mathcal{E}^*$. The dropout rate of the emulator is denoted as $\beta = \frac{|L_{\mathcal{E}}|}{n_{\mathcal{E}^*}}$. For convenience, we call $\mathcal{E}$ as emulator and $\mathcal{E}^*$ as non-compressed emulator. Let $\boldsymbol{w}_{\mathcal{E}}$ and $\boldsymbol{w}_{\mathcal{E}^*}$ be the parameters of $\mathcal{E}$ and $\mathcal{E}^*$, respectively.

After training, we can attain two combined models, namely, Adaptor + Emulator (AdapEmu, i.e., $\mathcal{E} \circ \mathcal{A}$), and Adaptor + Full (AdapFu, i.e., $\mathcal{E}^* \circ \mathcal{A}$). As Xiao et al. [43] describes, AdapFu performs better than AdapEmu. These two models have different functionalities in real-world scenarios: AdapEmu is adopted if the input contains sensitive information that cannot be shared with the LLM owner, e.g., drafting a petition letter, while AdapFu is adopted when the users aim to have better generation results, e.g., solving a math problem.

[1]The bottom/last layers refer to the transformer decoders near the output, while the top/first layers refer to the part close to the input.

Algorithm 1 LayerExtract

Input: pre-trained LLM $\mathcal{M}$ (layer index starts from 0), adapter size s, dropout rate β.

1: Get the size of model: $n \leftarrow |\mathcal{M}|$
2: Compute the number of layers in the compressed model, i.e.,
$$n' \leftarrow \lfloor \beta(n-s) \rfloor$$
3: Initialize non-compressed emulator $\mathcal{E}^* \leftarrow \{\mathcal{M}_0, \ldots \mathcal{M}_{n-s-1}\}$, adapter $\mathcal{A} \leftarrow \{\mathcal{M}_{n-s}, \ldots, \mathcal{M}_{n-1}\}$
4: Initialize emulator $\mathcal{E} \leftarrow \{\}$
5: Compute $stride \leftarrow (n-s-1)/(n'-1)$
6: **for** $j = 0, \ldots, n'-1$ **do**
7: Append $\mathcal{M}_{\lfloor j \times stride \rfloor}$ to emulator, i.e.,
$$\mathcal{E} \leftarrow \mathcal{E} \cup \{\mathcal{M}_{\lfloor j \times stride \rfloor}\}$$
8: **end for**
9: **return** $\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}, \boldsymbol{w}_{\mathcal{E}^*}$

Step 3: Pre-alignment. Before the FL training stage, we pre-align the emulator with the non-compressed one such that it can mimic the performance of the raw model. Assume there is a public dataset $\mathcal{D}_{public}$ available on the server, which consists of a bunch of data $(\boldsymbol{x}, \boldsymbol{y})$, representing the input and the ground truth, respectively. Therefore, in the rest of the section, we assume the input $\boldsymbol{x}$ contains an attention mask that can identify the ground truth.

Instead of training the compressed model with the ground truth, we utilize knowledge distillation [10] to transfer the general linguistic patterns from the original LLM to the compressed one by tuning the emulator $\mathcal{E}$. In specific, we ensure the emulator generates representations that have subtle differences from the non-compressed emulator with the given input on the ground truth part. To this end, we aim to minimize the following ℓ_2-norm:

$$\mathcal{L}_{repr} = \left\| \mathcal{E}\left(\boldsymbol{x}; \boldsymbol{w}_{\mathcal{E}}\right) - \mathcal{E}^*\left(\boldsymbol{x}; \boldsymbol{w}_{\mathcal{E}^*}\right) \right\|_2^2 \tag{3}$$

Additionally, we ensure the compressed model has consistent final outputs of the original LLM on the ground truth by minimizing the following KL divergence:

$$\mathcal{L}_{kd} = D_{KL}\left(\mathcal{M}(\boldsymbol{x}; \{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}^*}\}) \| \mathcal{M}(\boldsymbol{x}; \{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}\})\right) \tag{4}$$

In a nutshell, we optimize the emulator $\mathcal{E}$ by finding the optimal parameters for the following equation on the public dataset

$$\min_{\boldsymbol{w}_{\mathcal{E}}} \frac{1}{|\mathcal{D}_{public}|} \sum_{\boldsymbol{x} \in \mathcal{D}_{public}} \mathcal{L}_{repr} + \lambda \mathcal{L}_{kd} \tag{5}$$

Let the optimal emulator $\mathcal{E}$ for Equation (5) be $\mathcal{E}_{init}$ with the parameter of $\boldsymbol{w}_{\mathcal{E}_{init}}$. Denote the selected adapter $\mathcal{A}$ with the parameter of $\boldsymbol{w}_{\mathcal{A}_{init}}$. To this end, we distribute a compressed model to the clients with the initial parameters of $\{\boldsymbol{w}_{\mathcal{A}_{init}}, \boldsymbol{w}_{\mathcal{E}_{init}}\}$. To reduce the computation and communication costs, we incorporate LoRA [11] for the adapter $\mathcal{A}$ and the emulator $\mathcal{E}$, denoted as $\mathcal{A}_{lora}$ and $\mathcal{E}_{lora}$, respectively.

Before diving into the details of the proposed FedBiOT, we briefly go through the workflow as described in Figure 1. The figure visually presents the workflow of the federated learning process of our

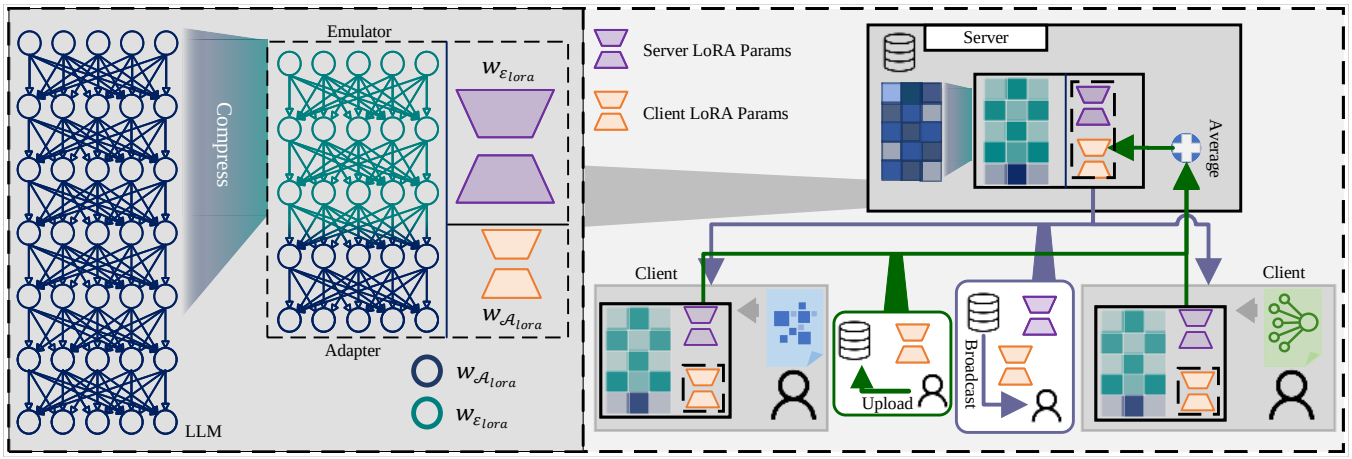


Figure 1: The Workflow of FedBiOT during the FL training

proposed FedBiOT, including the local updates on clients (Section 3.3) and the aggregation on the server (Section 3.4). At the beginning of clients' fine-tuning, the server broadcasts the adapter $\mathcal{A}_{lora}$ and the emulator $\mathcal{E}_{lora}$ to the clients. Subsequently, the clients perform multiple local updates to fine-tune the adapter $\mathcal{A}_{lora}$ with their local datasets. After the local updates, the client uploads the adapter $\mathcal{A}_{lora}$ to the server, and the server thereby aggregates the adapters. To ensure that the emulator is still able to reproduce the behavior of the uncompressed LLM, the server fine-tunes the emulator $\mathcal{E}_{lora}$ with the public dataset. Finally, the server distributes the updated parameters to the clients and launches a new round of training.

3.2 Formulation of Bi-level Optimization

As discussed in Section 3.1, we compress and divide the LLM into two parts, namely, an adapter and an emulator. These two components are designated to satisfy the following objectives:

- **Emulator** should be tuned towards perfectly imitating the non-compressed part in the full model, especially in extracting and encoding information on the server's datasets.
- **Adapter** should be able to digest the output of the emulator efficiently and should be encoded with the knowledge from clients' datasets effectively.

Define $\boldsymbol{w}_{\mathcal{A}} = \{\boldsymbol{w}_{\mathcal{A}_{init}}, \boldsymbol{w}_{\mathcal{A}_{lora}}\}$, $\boldsymbol{w}_{\mathcal{E}} = \{\boldsymbol{w}_{\mathcal{E}_{init}}, \boldsymbol{w}_{\mathcal{E}_{lora}}\}$ to integrate the LoRA parameters while the initial parameters for the adapter and the emulator remain unchanged during the training. Toward the goal, we formulate the objectives as a bi-level optimization problem:

$$\min_{\boldsymbol{w}_{\mathcal{A}_{lora}}} \sum_{m\in[M]} p_m F_m\left(\{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}\}\right) + \frac{\epsilon}{2}\left\|\hat{\boldsymbol{w}}_{\mathcal{A}_{lora}} - \hat{\boldsymbol{w}}^{(t)}_{\mathcal{A}_{lora}}\right\|_2^2 \tag{6}$$

$$s.t. \quad \boldsymbol{w}_{\mathcal{E}_{lora}} \in \arg\min_{\boldsymbol{w}_{\mathcal{E}_{lora}}} \frac{1}{|\mathcal{D}_{public}|} \sum_{\boldsymbol{x}\in\mathcal{D}_{public}} \mathcal{L}(\boldsymbol{x}),$$

$$\mathcal{L}(\boldsymbol{x}) \triangleq \left\|\mathcal{E}\left(\boldsymbol{x}; \boldsymbol{w}_{\mathcal{E}}\right) - \mathcal{E}^*\left(\boldsymbol{x}; \boldsymbol{w}_{\mathcal{E}^*}\right)\right\|_2^2 + \lambda \cdot D_{KL}\left(\mathcal{M}(\boldsymbol{x}; \{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}^*}\}) \| \mathcal{M}(\boldsymbol{x}; \{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}\})\right) \tag{7}$$

where $\boldsymbol{w}^{(t)}_{\mathcal{A}_{lora}}$ is the adapter LoRA received at the beginning of each communication round, $\hat{\boldsymbol{w}}_{\mathcal{A}_{lora}}$ reconstructs for the same size of the adapter $\boldsymbol{w}_{\mathcal{A}}$. $\mathcal{D}_{public}$ represents the public dataset on the server, which can be unlabeled. $D_{KL}(\cdot\|\cdot)$ is the KL divergence between two logits. ϵ and λ are hyperparameters.

The upper-level objective (Equation* (6)*). The upper-level objective function consists of two terms. The first term represents the loss of the model on local clients' data, with the current emulator and adapter. It follows a classic weighted average loss in FL to balance the loss of different clients' heterogeneous local data. The goal of introducing this term is straightforward: by minimizing the loss of the first term, we expect the emulator-adapter combination to be improved on the local training set. The second term is a regularization of the adapter component to ensure it will be within a reasonable distance from the synchronized and broadcast adapter at the beginning of each communication round. Enforcing a restriction on the adapter's change can reduce the difference of losses for the emulator distillation after locally adapter are tuned locally on clients, so it can help the convergence of emulator distillation.

The lower-level objective (Equation* (7)*). The first term in the constraint is the ℓ_2-norm difference between the activation output by the emulator and the full model. The second term is the KL divergence between the output of output distribution of the full model-adapter combination and the emulator-adapter. Although only the emulator is trainable to minimize the loss of these two terms, these two terms provide different optimization meaning for the emulator. The first term encourages the emulator to provide activations as close as possible to the full model, *excluding* the effect of the adapter. The second term ensures the emulator can provide output distributions close to the one when the full model with adapters is added on.

Discussion. The introduced algorithm can optimize the bi-level problems (i.e., Equation (6) and (7)) to an equilibrium point for both adapter and emulator. This is because when we optimize the adapter, the fixed emulator constrains its updates, and vice versa,

Algorithm 2 FedBiOT

Input: learning rate η, local updates K, global model alignment steps E, strength of full model alignment λ, local update regularization ϵ, total communication rounds R, pre-trained LLM $\mathcal{M}$ with parameter $\boldsymbol{w}$, adapter size s, dropout rate β, number of clients M.

$\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}, \boldsymbol{w}_{\mathcal{E}^*} \leftarrow \text{LayerExtract}(\mathcal{M}, s, \beta)$ ▷ See Algo. 1 for details
for $t = 0, \ldots, R-1$ **do**
  for $e = 0, \ldots, E-1$ **do**
    Randomly sample $(\boldsymbol{x}, \boldsymbol{y})$ from the public dataset $\mathcal{D}_{public}$
    Optimize $\boldsymbol{w}_{\mathcal{E}_{lora}}$ with respect to Equation (7)
  end for
  Communicate $\{\boldsymbol{w}_{\mathcal{A}_{lora}}, \boldsymbol{w}_{\mathcal{E}_{lora}}\}$ with clients $m \in [M]$
  for $m \in [M]$ **in parallel do**
    Initialize $\boldsymbol{w}^{(t)}_{\mathcal{A}_{lora}}$, $\boldsymbol{w}_{\mathcal{A}_{lora},m}$, and $\boldsymbol{w}_{\mathcal{E}}$ using Equation (8)
    for $k = 0, \ldots, K-1$ **do**
      Compute a gradient g using Equation (9)
      Update local model $\boldsymbol{w}_{\mathcal{A}_{lora},m}$ using Equation (10)
    end for
    Communicate $\boldsymbol{w}_{\mathcal{A}_{lora},m}$ with the server
  end for
  Update $\boldsymbol{w}_{\mathcal{A}_{lora}}$ using Equation (11)
end for
return AdapEmu $\{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}\}$, AdapFu $\{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}^*}\}$

and thereby, the emulator and adapter are distilled or trained interchangeably. At this equilibrium, the emulator can more faithfully extract and encode the information for the clients' dataset and benefit from the training of the adapter in reverse.

Additionally, FedBiOT does not require the design of an emulator to follow linear dropout. Instead, this is a general framework that compresses an LLM and divides it into two components: an emulator and an adapter. There are numerous designs for the emulator, but they share the same objective where the emulator simulates the non-compressed part of an LLM. For simplicity, we follow offsite-tuning [43] and prepare the emulator by means of uniform layer dropout [26] to demonstrate the effectiveness of FedBiOT.

3.3 Client Updates

During the local updates, the clients barely fine-tune the parameters of the adapter $\mathcal{A}$ while fixing the parameters of the emulator $\mathcal{E}$. By enabling LoRA, the LoRA of the adapter will get updated, and therefore, the clients should upload the updated $\boldsymbol{w}_{\mathcal{A}_{lora}}$ to the server after the local fine-tuning ends.

Consider client $i \in [M]$ performs the local updates at t-th round. Before optimizing the adapter locally, the client receives the updated emulator $\boldsymbol{w}_{\mathcal{E}_{lora}}$ and adapter $\boldsymbol{w}_{\mathcal{A}_{lora}}$ from the client, and we denote them by

$$\boldsymbol{w}^{(t)}_{\mathcal{A}_{lora}} \leftarrow \boldsymbol{w}_{\mathcal{A}_{lora}}, \quad \boldsymbol{w}_{\mathcal{A}_{lora},m} \leftarrow \boldsymbol{w}_{\mathcal{A}_{lora}}, \quad \boldsymbol{w}_{\mathcal{E}} \leftarrow \{\boldsymbol{w}_{\mathcal{E}_{init}}, \boldsymbol{w}_{\mathcal{E}_{lora}}\} \tag{8}$$

Suppose the client performs the local update for K times. In each local update, we solely optimize $\boldsymbol{w}_{\mathcal{A}_{lora},m}$, a LoRA module of the adapter. Therefore, based on Equation (6), the gradient w.r.t. $\boldsymbol{w}_{\mathcal{A}_{lora},m}$ should be

$$g \leftarrow \nabla_{\boldsymbol{w}_{\mathcal{A}_{lora},m}} F_m\left(\{\boldsymbol{w}_{\mathcal{A}}, \boldsymbol{w}_{\mathcal{E}}\}\right) + \epsilon \left(\hat{\boldsymbol{w}}_{\mathcal{A}_{lora},m} - \hat{\boldsymbol{w}}^{(t)}_{\mathcal{A}_{lora}}\right)^T \left(\frac{\partial \hat{\boldsymbol{w}}_{\mathcal{A}_{lora},m}}{\partial \boldsymbol{w}_{\mathcal{A}_{lora},m}}\right) \tag{9}$$

where $\boldsymbol{w}_{\mathcal{A}} = \{\boldsymbol{w}_{\mathcal{A}_{init}}, \boldsymbol{w}_{\mathcal{A}_{lora},m}\}$ in the above formula. Let $\text{Optim}()$ be the optimizer (e.g., SGD and AdamW [21]) that updates the model parameters, and η be the learning rate. Therefore, in each local update, the local model is updated for

$$\boldsymbol{w}_{\mathcal{A}_{lora},m} \leftarrow \text{Optim}(\boldsymbol{w}_{\mathcal{A}_{lora},m}, g, \eta) \tag{10}$$

After finishing the local update, the client i sends $\boldsymbol{w}_{\mathcal{A}_{lora},m}$ to the server.

3.4 Model Aggregation

During the server aggregation, the server performs the weighted average to update the adapters $\mathcal{A}$ and fine-tune the emulator $\mathcal{E}$. By enabling the LoRA, only the parameters $\boldsymbol{w}_{\mathcal{A}_{lora}}$ in the adapter and $\boldsymbol{w}_{\mathcal{E}_{lora}}$ in the emulator are updated, while the rest (i.e., $\boldsymbol{w}_{\mathcal{A}_{init}}$ and $\boldsymbol{w}_{\mathcal{E}_{init}}$) remain unchanged.

First, the server collects a set of updated LoRAs of the adapter, i.e., $\left\{\boldsymbol{w}_{\mathcal{A}_{lora},m}\right\}_{m\in[M]}$ from the clients. Based on the definition of Equation (6), the server performs weighted aggregation via

$$\boldsymbol{w}_{\mathcal{A}_{lora}} \leftarrow \sum_{m\in[M]} p_m \boldsymbol{w}_{\mathcal{A}_{lora},m} \tag{11}$$

After the weighted averaging, the server distills the emulator $\mathcal{E}$ from the non-compressed $\mathcal{E}^*$ and the updated adapter $\mathcal{A}$ using the public dataset. Therefore, we fine-tune the emulator following Equation (7), which updates the LoRA of the emulator $\boldsymbol{w}_{\mathcal{E}_{lora}}$.

4 Experiments

4.1 Experimental Setup

This section discusses the implementation of our experiments, covering details such as the model utilized and evaluation metrics. The code is now available at https://github.com/HarliWu/FedBiOT.

Model and computation environment. The experiments utilize LLaMA-2-7B, an open-source pre-trained LLM maintained by Meta and released in July 2023 [34]. Preceding this, the model's first generation was introduced in February 2023 [33]. This model supports a maximum of 4096 input tokens and consists of 32 hidden layers with a total of 6.7 billion parameters. The experimental setup involves machines equipped with Nvidia A100 GPU cards, Intel Xeon Platinum 8369B CPUs, and a 512GB RAM configuration.

Datasets and Tasks. In the experiments, we use the benchmark datasets and tasks in [13] to train and evaluate the LLM on three different NLP tasks, covering math problem-solving, code generation, and question-answering:

- For **math problem-solving**, we split the GSM-8K training dataset [5] ensuring i.i.d. across three clients, and we assess the updated model using the GSM-8K test dataset.
- For **code generation**, we fine-tune the model with the Rosetta dataset [3], which is partitioned across the programming languages, and a total of nine clients separately hold the data from nine different programming languages. Regarding its evaluation, we utilize HumanEvalX [50], an extension of a coding evaluation

Table 1: Dataset details for LLM training and evaluation

Task	Training Dataset	# training samples	# clients	Partition Rules	Max.	Min.	Std.	Test Dataset	# test samples
Math Problem Solving	GSM-8K	7473	3	i.i.d.	2491	2491	0	GSM-8K	1319
Code Generation	Rosetta	7954	9	Prog. Lang.	1172	439	236.94	HumanEvalX	656
Question Answering	Dolly	15015	8	Category	3611	711	795.06	Helm	NA
Public Dataset	Alpaca	52002		————————————————————					

dataset [4] that requires the model to fill in the code for a given problem in the required programming language (i.e., C++, GO, Java, Python).

- For **question answering**, the model is trained on dolly-15K [6], which is partitioned into 8 clients based on the categories of the questions, and we evaluate the new model with the selected tasks on HELM [17].

Table 1 gives a detailed description of these three tasks. As Section 3 mentions, the server will perform the emulator alignment during the model aggregation. Then, we use the Alpaca dataset [31] as the public dataset for the server to do the emulator alignment for all three NLP tasks.

Implementation. This work is built upon an open-source federated learning platform named FederatedScope [44]. The training data are reformatted following the predesigned instructions [3, 48].

Different from [13, 43], we regard the last two and the last four decoders as the adapter. The experiments consider two dropout rates, i.e., $\beta \in \{0.2, 0.5\}$, and we obtain the emulators with layer dropout following Xiao et al. [43]. Without special annotation, we use the following local training setting: in each communication round, each client performs 30 local updates, and the batch size of every local update is 10. Before launching the FL training, we fine-tune the emulator for 500 iterations to generate a distilled emulator $\mathcal{E}$ towards minimizing the loss of Equation (7). During the FL training, the server takes 10 iterations to align the emulator $\mathcal{E}$ with $\mathcal{E}^*$ between two successive communication rounds after aggregating local adapters with FedAvg [15]. These experiments run for 500 communication rounds, and we report the results based on the fine-tuned LLM obtained at the 500th round. During the training, we only fine-tune the adapter in the clients' local update procedures, and we update the emulator on the server side. In other words, other parts of the pre-trained model, such as word embeddings, are frozen during the training.

LoRA, Optimizers and Hyperparameters. We add the LoRA to all decoder layers in the adapter and the emulator by setting the rank to 8 and the alpha to 16. We use AdamW as an optimizer to solve Equation (6) and (7) on the clients (for the adapters) and the server (for the emulators), respectively. We search for the best learning rate in $\{1 \times 10^{-5}, 3 \times 10^{-5}, 5 \times 10^{-5}, 8 \times 10^{-5}, 1 \times 10^{-4}\}$. We set the momentum for $(0.9, 0.95)$. As for other hyperparameters related to the optimizer, we use the default setting. Furthermore, we also conduct grid search for FedBiOT-specific hyperparameters, i.e., ϵ and λ. Throughout the experiments, we demonstrate the result of the best hyperparameter combination. To avoid randomness, we utilize three different random seeds and report the averaged results.

Table 2: Test accuracy on math problem-solving task under different dropout rates

Dropout Rate (β)	Methods	AdapEmu	AdapFu
$\beta = 0.0$	Few-shot CoT	NA	13.42% (177/1319)
$\beta = 0.2$	Offsite-tuning	3.03% (40/1319)	9.93% (131/1319)
	FedOT	2.43% (32/1319)	10.16% (134/1319)
	FedBiOT (Adapter 2)	**3.71% (49/1319)**	15.16% (200/1319)
	FedBiOT (Adapter 4)	3.41% (45/1319)	**15.23% (201/1319)**
$\beta = 0.5$	Offsite-tuning	**2.27% (30/1319)**	7.58% (100/1319)
	FedOT	1.90% (25/1319)	7.51% (99/1319)
	FedBiOT (Adapter 2)	2.05% (27/1319)	11.83% (156/1319)
	FedBiOT (Adapter 4)	1.82% (24/1319)	**14.03% (185/1319)**

Baselines. Offsite-tuning is the only method that satisfies the constraints that fine-tuning without access to full model. Xiao et al. [43] introduces a single-client offsite-tuning, while Kuang et al. [13] extends it to an FL version (i.e., FedOT). We apply offsite-tuning with one single client, where all data are loaded to the client. As FedOT supports FL, we reproduce the algorithm to work on the FL tasks. In terms of the setting of the adapters and the emulators, both Offsite-tuning and FedOT treat the first two and the last two decoders as the adapter. To enable the parameter-efficient fine-tuning for both baselines, we add LoRA to both baselines, the same as the setting adopted by FedBiOT.

Evaluation Metric. In the experiments, we report the results on two models, i.e., AdapEmu and AdapFu, as defined in Section 3.1. The evaluation metrics for each task follow Kuang et al. [13], and the detailed description is given in Appendix A.

4.2 Quantitative Evaluation on i.i.d. Data

We demonstrate the experimental results of GSM-8K provided in Table 2 and highlight the worth-noted phenomenon when the data are i.i.d. across the clients.

A notable phenomenon observed in the table is that AdapEmu significantly falls behind AdapFu, particularly at a low dropout rate (i.e., $\beta = 0.2$). To explain this, we examine the accuracy of the LLaMA-2 model with a dropout rate of 0.2, which is 2.12% without fine-tuning and increases to 2.43% after fine-tuning the emulator with a public dataset. The performance gap between AdapEmu and AdapFu can be attributed to layer dropout, which reduces the size of the LLM and subsequently impacts its performance. Additionally, this result highlights the difficulty of accurately reproducing the non-compressed parts with the emulator. Fortunately, all methods

Table 3: Pass@1 (%) and Pass@10 (%) in code generation task at various rounds when dropout rate is 0.2

Method	Model	C++		GO		Java		Python	
		Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10
Offsite-tuning	AdapEmu	3.99	*6.45*	1.80	2.44	5.64	6.09	5.01	6.38
	AdapFu	8.78	10.82	4.94	6.63	9.57	12.81	13.19	17.32
FedOT	AdapEmu	2.50	4.89	1.86	3.05	5.00	5.49	4.91	6.83
	AdapFu	8.60	11.36	5.95	7.11	6.30	9.42	12.23	13.58
FedBiOT (Adapter 2)	AdapEmu	*4.82*	6.43	*3.57*	*4.85*	*5.92*	*6.36*	4.97	*6.95*
	AdapFu	**9.76**	**14.18**	**9.97**	**13.29**	**12.93**	**16.28**	**14.91**	**19.77**
FedBiOT (Adapter 4)	AdapEmu	3.20	4.57	2.20	2.44	4.91	5.73	*5.43*	6.10
	AdapFu	9.12	13.41	8.02	11.08	11.28	13.10	14.57	18.41

Table 4: Pass@1 (%) and Pass@10 (%) in code generation task at various rounds when dropout rate is 0.5. We do not show AdapEmu's performance because it struggles to generate meaningful codes, accounting for its small size.

Method	Model	C++		GO		Java		Python	
		Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10	Pass@1	Pass@10
Offsite-tuning	AdapFu	5.30	7.26	3.32	7.55	4.61	5.33	8.75	10.26
FedOT	AdapFu	4.92	7.33	5.00	8.33	3.86	4.37	7.33	8.91
FedBiOT (Adapter 2)	AdapFu	**7.71**	**11.84**	**7.68**	**10.01**	**9.51**	**14.34**	13.29	**16.87**
FedBiOT (Adapter 4)	AdapFu	5.03	11.09	6.25	8.47	7.41	13.32	**13.54**	16.74

improve AdapEmu's performance compared to the version without fine-tuning.

When we take a look at the proposed FedBiOT at different adapters' sizes, we notice that FedBiOT with adapter 4 achieves better performance than that with adapter 2 under the AdapFu setting. As we know, a larger adapter has more trainable parameters, and therefore, it can easily absorb the knowledge from the downstream tasks. Note that the performances of these two adapter settings have subtle differences under AdapEmu, meaning that their emulator achieves very similar effects to the non-compressed emulator. When we plug the adaptor back into the non-compressed emulator, the adapter with more trainable parameters obviously can achieve a better performance.

When comparing our proposed model with the baselines, we can notice a significant dominance in performance, especially in the AdapFu setting. More specifically, when the dropout rate becomes larger, the performance of AdapFu with FedBiOT decreases more mildly in contrast to other baselines. This is thanks to two factors: 1) the regularization term ensures the adapters will not change dramatically; 2) the on-the-fly distillation of the emulator with mixed losses can work better with clients' data. Although the other two baselines use a public dataset to achieve similar functionality, the deterioration may still occur due to the data domain shift and the significant information loss.

4.3 Quantitative Evaluation on non-i.i.d. Data

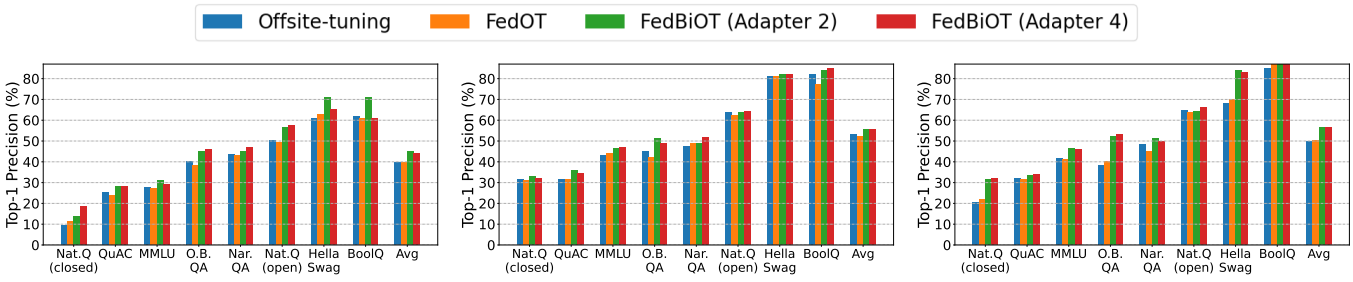
According to Table 1, code generation and question answering are two tasks split in non-i.i.d. styles. In this section, we evaluate our proposed FedBiOT when it trains an LLM with a non-i.i.d. dataset. It is worth noting that the evaluation task could be either in-distribution or out-of-distribution to the training dataset.

Code generation. Table 3 and 4 illustrate the best results in different programming languages based on different hyperparameter settings. Let us take a look at the results of the FedBiOT at different adapter sizes. Apparently, FedBiOT with two layers of adapter constantly outperforms FedBiOT with four under both AdapEmu and AdapFu. This conclusion is different from the one when an LLM is trained with an i.i.d. dataset. The discrepancy can be attributed to the clients' objectives: under i.i.d. datasets, a larger adapter size benefits training by absorbing downstream linguistic patterns uniformly. Conversely, with non-i.i.d. datasets, clients are biased towards their local optima, where the emulator's effect becomes crucial.

When comparing our proposed algorithm with the baselines, we notice a distinct dominance in AdapFu across all programming languages. In particular, when the dropout rate is 0.5, we can achieve up to 6% improvement over other baselines in terms of Pass@1, and up to 10% improvement of Pass@10. Notably, the most distinct dominance can be witnessed under the "column" of Java in Table 4.

Question Answering. Figure 2 shows the evaluation results using the HELM benchmark while we train the LLM with Dolly-15K. Generally speaking, FedBiOT (Adapter 2) performs significantly better than Adapter 4 in some tasks in terms of AdapEmu. As both AdapEmu have the same number of layers, this result exhibits the importance of the emulator, i.e., the model with a larger emulator can achieve leading performance. To some extent, this result supports our previous conclusion that an emulator plays a more important role than an adapter in a non-i.i.d. task. As for AdapFu, the performance difference is trivial between the two adapter sizes.

The proposed algorithm outperforms offsite-tuning and FedOT in most datasets, which is consistent with the findings in other training tasks. The dominance of AdapFu becomes more pronounced as



(a) AdapEmu (Dropout rate 0.2) (b) AdapFu (Dropout rate 0.2) (c) AdapFu (Dropout rate 0.5)

Figure 2: Test accuracy in eight types of question-answering tasks (Left to right: Natural Questions (closed-book), QuAC, MMLU, OpenbookQA, NarrativeQA, Natural Questions (open-book), HellaSwag, BoolQ) and the average accuracy under different baselines (bars from left to right: Offsite-tuning, FedOT, FedBiOT (Adapter 2), FedBiOT (Adapter 4)) and different dropout rates.

the dropout rate increases from 0.2 to 0.5. For instance, FedBiOT is approximately 10% better than the baselines at a 0.5 dropout rate in Natural Questions (closed-book), compared to a 2% improvement at a 0.2 dropout rate. Notably, comparing Figure 2b and 2c, we notice that FedBiOT is mildly affected by changes in the dropout rate, while the baselines suffer significant degradation as the dropout rate increases. This stability can be attributed to round-by-round emulator alignment, where the non-compressed part of the full model is set as an anchor, regardless of the dropout rate. Consequently, this approach stabilizes the adapter training process, ensuring that adapters of the same size achieve similar performance across varying dropout rates.

4.4 Discussion on Computation and Communication Overhead

Table 5 presents the computation and communication overhead of different methods under different dropout rates. As mentioned in the experimental setting, all algorithms have been applied with LoRA, and therefore, the number of trainable parameters dramatically reduces. From the clients' perspectives, the number of trainable parameters is determined by the number of decoder layers in the adapter. Apparently, FedBiOT (Adapter 2) should be with the minimum number of trainable parameters among other methods.

The computation costs in Table 5 are measured by per-token floating point operation (FLOP/token). As we can see, the proposed FedBiOT costs less overhead than offsite-tuning and FedOT. The difference arises on account of the position of the trainable parameters. The adapter of the proposed FedBiOT is near the output layer. As for offsite-tuning and FedOT, the adapters are located separately at the top and the bottom two layers, thereby consuming more computation costs in the backward propagation for transmitting the derivative from the bottom to the top.

However, our proposed method may require more communication overhead than the baselines. This is because the server should transmit the LoRA parameters of both the adapter and the emulator to the clients in our proposed method, while in offsite-tuning and FedOT, the server merely transmits the aggregated LoRA of the adapter to the clients. However, the overall cost is trivial, compared to the full LLM transmission at a cost of 28GB.

Table 5: Computation and communication costs of different methods under different dropout rates at client side.

Dropout Rate (β)	Methods	#Layers in Adapter	#Layers in Emulator	Trainable Param. (M)	Comp. Costs (GFLOP/token)	Comm. Costs (MB/round)
$\beta = 0.2$	Offsite-tuning /FedOT	4	22	0.524	10.33	4.19
	FedBiOT (Adapter 2)	2	24	0.262	5.47	14.68
	FedBiOT (Adapter 4)	4	22	0.524	5.87	15.73
$\beta = 0.5$	Offsite-tuning /FedOT	4	14	0.524	7.09	4.19
	FedBiOT (Adapter 2)	2	15	0.262	3.65	9.96
	FedBiOT (Adapter 4)	4	14	0.524	4.25	11.53

5 Conclusion

In this paper, we introduce FedBiOT, a federated learning algorithm that avoids full model fine-tuning while substantially reducing computation overhead. Specifically, we compress the LLM and divide it into two components, namely, an emulator and an adapter. By formulating a bi-level optimization problem, our proposed FedBiOT ensures that the emulator partially simulates the original LLM, while the adapter focuses on learning domain-specific linguistic patterns. Extensive experiments show the superiority of the proposed FedBiOT working with LLaMA-2, where it can achieve significant accuracy improvement than the existing baselines (i.e., Offsite-tuning and FedOT) in all tasks (i.e., math problem-solving, code generation, and question answering).

Acknowledgments

The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by the US National Science Foundation under grants NSF-IIS 1747614 and NSF-IIS 2141037. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Durmus Alp Emre Acar, Yue Zhao, Ramon Matas, Matthew Mattina, Paul Whatmough, and Venkatesh Saligrama. 2020. Federated Learning Based on Dynamic

Regularization. In *Proc. of International Conference on Learning Representations (ICLR'20)*.
[2] CCPA. 2023. *California Consumer Privacy Act (CCPA)*. https://oag.ca.gov/privacy/ccpa
[3] Sahil Chaudhary. 2023. Code Alpaca: An Instruction-following LLaMA model for code generation. https://github.com/sahil280114/codealpaca.
[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[5] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).
[6] Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. *Free Dolly: Introducing the World's First Truly Open Instruction-Tuned LLM*. https://www.databricks.com/blog/2023/04/12/dolly-first-open-commercially-viable-instruction-tuned-llm
[7] Jiaxi Cui, Zongjian Li, Yang Yan, Bohua Chen, and Li Yuan. 2023. Chatlaw: Open-source legal large language model with integrated external knowledge bases. *arXiv preprint arXiv:2306.16092* (2023).
[8] GDPR. 2016. *Regulation (EU) 2016/679 of the European Parliament and of the Council*. https://data.europa.eu/eli/reg/2016/679/oj
[9] Shiqi He, Qifan Yan, Feijie Wu, Lanjun Wang, Mathias Lécuyer, and Ivan Beschastnikh. 2023. GlueFL: Reconciling Client Sampling and Model Masking for Bandwidth Efficient Federated Learning. *Proc. of Machine Learning and Systems (MLSys'23)*.
[10] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).
[11] Edward J Hu, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2021. LoRA: Low-Rank Adaptation of Large Language Models. In *Proc. of International Conference on Learning Representations (ICLR'21)*.
[12] Sai Praneeth Karimireddy, Satyen Kale, Mehryar Mohri, Sashank Reddi, Sebastian Stich, and Ananda Theertha Suresh. 2020. Scaffold: Stochastic controlled averaging for federated learning. In *Proc. of International conference on machine learning (ICML'20)*. 5132–5143.
[13] Weirui Kuang, Bingchen Qian, Zitao Li, Daoyuan Chen, Dawei Gao, Xuchen Pan, Yuexiang Xie, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. FederatedScope-LLM: A Comprehensive Package for Fine-tuning Large Language Models in Federated Learning. In *Proc. of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'24)*.
[14] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The Power of Scale for Parameter-Efficient Prompt Tuning. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP'21)*. 3045–3059.
[15] Xiang Li, Kaixuan Huang, Wenhao Yang, Shusen Wang, and Zhihua Zhang. 2019. On the Convergence of FedAvg on Non-IID Data. In *Proc. of International Conference on Learning Representations (ICLR'19)*.
[16] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *Proc. of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing (ACL/IJNLP'21)*. 4582–4597.
[17] Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. 2022. Holistic evaluation of language models. *arXiv preprint arXiv:2211.09110* (2022).
[18] Tao Lin, Lingjing Kong, Sebastian U Stich, and Martin Jaggi. 2020. Ensemble distillation for robust model fusion in federated learning. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS'20)*. 2351–2363.
[19] Zihao Lin, Yan Sun, Yifan Shi, Xueqian Wang, Lifu Huang, Li Shen, and Dacheng Tao. 2023. Efficient federated prompt tuning for black-box large pre-trained models. *arXiv preprint arXiv:2310.03123* (2023).
[20] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. 2023. GPT understands, too. *AI Open* (2023).
[21] Ilya Loshchilov and Frank Hutter. 2018. Decoupled Weight Decay Regularization. In *Proc. of International Conference on Learning Representations (ICLR'18)*.
[22] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Proc. of Artificial intelligence and statistics (AISTAT'17)*. 1273–1282.
[23] John J Nay, David Karamardian, Sarah B Lawsky, Wenting Tao, Meghana Bhat, Raghav Jain, Aaron Travis Lee, Jonathan H Choi, and Jungo Kasai. 2024. Large language models as tax attorneys: a case study in legal capabilities emergence. *Philosophical Transactions of the Royal Society A* 382, 2270 (2024), 20230159.
[24] OpenAI. 2023. Fine-tuning - OpenAI API. https://platform.openai.com/docs/guides/fine-tuning. Accessed: 2023-09-29.
[25] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS'22)*. 27730–27744.
[26] Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. 2023. On the effect of dropping layers of pre-trained transformer models. *Computer Speech & Language* 77 (2023), 101429.
[27] Karan Singhal, Shekoofeh Azizi, Tao Tu, S Sara Mahdavi, Jason Wei, Hyung Won Chung, Nathan Scales, Ajay Tanwani, Heather Cole-Lewis, Stephen Pfohl, et al. 2023. Large language models encode clinical knowledge. *Nature* 620, 7972 (2023), 172–180.
[28] Alessandro Sordoni, Xingdi Yuan, Marc-Alexandre Côté, Matheus Pereira, Adam Trischler, Ziang Xiao, Arian Hosseini, Friederike Niedtner, and Nicolas Le Roux. 2023. Joint prompt optimization of stacked llms using variational inference. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS'23)*.
[29] Jingwei Sun, Ziyue Xu, Hongxu Yin, Dong Yang, Daguang Xu, Yiran Chen, and Holger R Roth. 2023. FedBPT: Efficient Federated Black-box Prompt Tuning for Large Language Models. *arXiv preprint arXiv:2310.01467* (2023).
[30] Youbang Sun, Zitao Li, Yaliang Li, and Bolin Ding. 2024. Improving LoRA in Privacy-preserving Federated Learning. In *Proc. of The International Conference on Learning Representations (ICLR'24)*.
[31] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
[32] Arun James Thirunavukarasu, Darren Shu Jeng Ting, Kabilan Elangovan, Laura Gutierrez, Ting Fang Tan, and Daniel Shu Wei Ting. 2023. Large language models in medicine. *Nature medicine* 29, 8 (2023), 1930–1940.
[33] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).
[34] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
[35] Haozhao Wang, Yichen Li, Wenchao Xu, Ruixuan Li, Yufeng Zhan, and Zhigang Zeng. 2023. Dafkd: Domain-aware federated knowledge distillation. In *Proc. of the IEEE/CVF conference on Computer Vision and Pattern Recognition (CVPR'23)*. 20412–20421.
[36] Haozhao Wang, Haoran Xu, Yichen Li, Yuan Xu, Ruixuan Li, and Tianwei Zhang. 2023. FedCDA: Federated Learning with Cross-rounds Divergence-aware Aggregation. In *Proc. of The International Conference on Learning Representations (ICLR'23)*.
[37] Haoyu Wang, Handong Zhao, Yaqing Wang, Tong Yu, Jiuxiang Gu, and Jing Gao. 2022. FedKC: Federated knowledge composition for multilingual natural language understanding. In *Proc. of the ACM Web Conference 2022 (WWW'22)*. 1839–1850.
[38] Jianyu Wang, Qinghua Liu, Hao Liang, Gauri Joshi, and H Vincent Poor. 2020. Tackling the objective inconsistency problem in heterogeneous federated optimization. In *Proc. of Advances in neural information processing systems (NeurIPS'20)*. 7611–7623.
[39] Sheng Wang, Zihao Zhao, Xi Ouyang, Qian Wang, and Dinggang Shen. 2023. Chatcad: Interactive computer-aided diagnosis on medical image using large language models. *arXiv preprint arXiv:2302.07257* (2023).
[40] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned Language Models are Zero-Shot Learners. In *Proc. of International Conference on Learning Representations (ICLR'21)*.
[41] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proc. of Advances in Neural Information Processing Systems (NeurIPS'22)*. 24824–24837.
[42] Feijie Wu, Song Guo, Zhihao Qu, Shiqi He, Ziming Liu, and Jing Gao. 2023. Anchor sampling for federated learning with partial client participation. In *Proc. of International Conference on Machine Learning (ICML'23)*. 37379–37416.
[43] Guangxuan Xiao, Ji Lin, and Song Han. 2023. Offsite-tuning: Transfer learning without full model. *arXiv preprint arXiv:2302.04870* (2023).
[44] Yuexiang Xie, Zhen Wang, Dawei Gao, Daoyuan Chen, Liuyi Yao, Weirui Kuang, Yaliang Li, Bolin Ding, and Jingren Zhou. 2023. FederatedScope: A Flexible Federated Learning Platform for Heterogeneity. In *Proc. of the VLDB Endowment (VLDB'23)*. 1059–1072.
[45] Liping Yi, Han Yu, Gang Wang, and Xiaoguang Liu. 2023. Fedlora: Model-heterogeneous personalized federated learning with lora tuning. *arXiv preprint arXiv:2310.13283* (2023).
[46] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks?. In *Proc. of Advances in neural information processing systems (NeurIPS'14)*.
[47] Jie Zhang, Song Guo, Xiaosong Ma, Haozhao Wang, Wenchao Xu, and Feijie Wu. 2021. Parameterized knowledge transfer for personalized federated learning. In

Proc. of Advances in Neural Information Processing Systems (NeurIPS'21). 10092–10104.

[48] Jianyi Zhang, Saeed Vahidian, Martin Kuo, Chunyuan Li, Ruiyi Zhang, Tong Yu, Guoyin Wang, and Yiran Chen. 2024. Towards building the federatedGPT: Federated instruction tuning. In *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'24)*. 6915–6919.

[49] Zhuo Zhang, Yuanhang Yang, Yong Dai, Qifan Wang, Yue Yu, Lizhen Qu, and Zenglin Xu. 2023. FedPETuning: When federated learning meets the parameter-efficient tuning methods of pre-trained language models. In *Proc. of Annual Meeting of the Association of Computational Linguistics (ACL'23)*. 9963–9977.

[50] Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proc. of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD'23)*. 5673–5684.

A Testing Dataset and Evaluation

As described in Table 1, we utilize three datasets to assess the fine-tuning performance. In this section, we briefly introduce all these datasets and provide the details about how they evaluate a given LLM.

GSM-8K.. We use the GSM-8K test set [5] to evaluate the ability of a large language model (LLM) to solve math problems. This dataset includes "questions" and "ground truth" answers. We assess correctness by determining how often the LLM answers a given question correctly. Following chain of thought (CoT) [41], we prepare a set of sample questions (a.k.a. few-shot prompting) and prompt the LLM to generate step-by-step solutions, ensuring the answers are formatted correctly. Finally, we extract the answers from these solutions and compare them with the ground truth to calculate the correctness rate.

HumanevalX.. This is a task for code autofill, which consists of 164 test samples for five programming languages [50]. It is worth noting that we use four of them (i.e., C++, GO, Java, and Python) because there are no JavaScript codes in the training dataset. Each test sample is constituted with "task id", "prompt" (i.e., Task description with partial codes), "entry point" (i.e., the function to be achieved), "canonical solution" (i.e., a sampled solution), and "test" (i.e., evaluate if the generated code can obtain the correct answer based on the given input). In this task, we use "prompt" as the input and generate five versions of codes using a given model. We compile the code and check if it can pass the given "test". Let c be the number of correct codes generated by LLM and passed unit tests, and therefore, Pass@k can be computed via

$$\text{Pass@k} = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]$$

HELM.. HELM [17] is a benchmark that contains a wide range of NLP tasks. We upload the well-trained models to the benchmark and evaluate them on question-answering tasks, which includes eight datasets, i.e., MMLU, BoolQ, NarrativeQA, Natural Questions (closed-book), Natural Questions (open-book), QuAC, HellaSwag, OpenbookQA. For different tasks, the results come from different metrics, i.e., **exact match** for *HellaSwag*, *OpenbookQA*, and *MMLU*; **quasi-exact match** for *BoolQ*; **F1** for the rest.