Menos: Split Fine-Tuning Large Language Models with Efficient GPU Memory Sharing

Chenghao Hu
ch.hu@mail.utoronto.ca
University of Toronto
Toronto, Ontario, Canada

Baochun Li
bli@ece.toronto.edu
University of Toronto
Toronto, Ontario, Canada

Abstract

Fine-tuning of pre-trained large language models has become increasingly popular, yet existing fine-tuning methods are typically centralized, requiring users to send local data to centralized servers, or model owners to open-source their models. However, data and models are valuable assets that few enterprises and users wish to share. In this paper, we deviate from conventional wisdom and advocate the use of split learning for fine-tuning models with private data, local to each of the clients. The most formidable challenge to split fine-tuning is the size of large language models: when multiple clients start their fine-tuning tasks, their use of GPU memory will overwhelm a GPU-equipped server, especially as the number of clients scales up. To address this challenge, we present *Menos*, the first memory-efficient split fine-tuning framework designed to optimize the server GPU footprint through spatial and temporal sharing. Specifically, *Menos* utilizes the adapter-based nature of modern fine-tuning techniques, and proposes to spatially share the base model parameters among multiple clients. It also schedules memory-intensive operations during the communication gaps of split learning, thereby temporally sharing limited GPU memory at runtime. Comprehensive real-world evaluations using state-of-the-art large language models demonstrate the effectiveness of *Menos*, reducing GPU memory consumption by up to 72%, yet incurring negligible overhead.

CCS Concepts: • **Computer systems organization** → **Client-server architectures**; • **Computing methodologies** → **Distributed algorithms**; **Neural networks**.

Keywords: Split Learning, Fine-Tuning, Cloud Computing, System Design

MIDDLEWARE '24, December 2–6, 2024, Hong Kong, Hong Kong

ACM ISBN 979-8-4007-0623-3/24/12…$15.00
https://doi.org/10.1145/3652892.3700758

ACM Reference Format:
Chenghao Hu and Baochun Li. 2024. *Menos:* Split Fine-Tuning Large Language Models with Efficient GPU Memory Sharing. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3652892.3700758

1 Introduction

In recent years, large pre-trained language models have become increasingly popular across a wide range of applications [35]. More and more users are employing model fine-tuning techniques to customize these models for their own specific purposes [28]. Fine-tuning allows users to adapt pre-trained models to perform well on downstream tasks by training on a smaller dataset relevant to the target domain. However, most existing fine-tuning approaches rely on centralized architectures. Users must send their local private data to a central server in order to fine-tune a model [29]. Alternatively, if users insist on carrying out the fine-tuning process locally, model owners must open-source their pre-trained parameters.

Unfortunately both scenarios are undesirable, as user data and model parameters are valuable assets that few enterprises and users wish to share. It is especially the case for user data, as they often contain sensitive or confidential information [18], putting user data under potential data privacy risks if that data is mishandled or breached. At the same time, commercially successful model owners have invested substantial resources into developing their pre-trained models like ChatGPT, and are reluctant to release their full model parameters publicly because doing so would allow competitors to freely use those models and undermine their own competitive advantage.

Split learning [7, 26, 34] offers a potential solution to this privacy dilemma of model fine-tuning. In split learning, the training process is distributed between the server and its clients. As the term readily suggests, the model itself is split, with one portion residing on the client and the remaining portion on the server. The server and clients iteratively exchange intermediate model activations and gradients to collaboratively update the model without directly sharing raw data or full model parameters. In this way, clients are able to maintain the privacy and confidentiality of their data, only sharing intermediate results. Meanwhile, model owners can keep a majority of the model parameters secret on the

server side. This enables a privacy-preserving approach to collaborative model fine-tuning.

However, since most of the model parameters and operations are hosted and conducted on the server, fine-tuning large pre-trained models introduces new challenges around server GPU memory consumption. State-of-the-art models like *Llama 2* [32] requires more than 20 GB of GPU memory to store the smallest version with 7 billion parameters, not including other states, such as activations and optimizer variables, that must be maintained during the fine-tuning process. As the number of clients grows, the total burden of GPU memory on the servers accumulates rapidly. With billions of parameters per model, supporting split fine-tuning across many clients can quickly exceed the memory capacity of even high-end GPUs, not to mention the exorbitant costs of operating these GPU servers to support split fine-tuning.

A variety of techniques have been proposed for reducing GPU memory consumption during model training that could be adapted to fine-tuning scenarios. For instances, model quantization [2–5] aims to compress the model footprint by quantizing parameters to lower bit depths. Other methods like gradient checkpointing [1, 15] and accumulation can decrease runtime memory for activations and gradients. However, these approaches do not consider the unique challenges of split learning across multiple clients. They optimize GPU memory for a single task, but do not address scalability when handling many model variants across clients.

In this paper, we introduce *Menos*, the first memory-efficient split fine-tuning framework optimizing server GPU footprint as the number of clients scales. Unlike existing methods, *Menos* identifies unique challenges and opportunities in combining split learning with model fine-tuning. In particular, it provides a tailored solution to address GPU memory limitations when the number of clients scales up. Therefore, existing methods are complementary to *Menos* and can be combined for further improvements. The original contributions of this work can be summarized as follows:

First, *Menos* utilizes the special feature of adapter-based model fine-tuning techniques [9, 10, 16] where only the adapter parameters are trained while the base model parameters are fixed. Adapters are lightweight neural network modules attached to a pre-trained model. Fine-tuning only the adapters while freezing the base model has been shown to achieve comparable performance to full model fine-tuning [9]. This allows *Menos* to spatially share the base model parameters among multiple clients. Such base model sharing mechanism allows *Menos* to conduct concurrent fine-tuning with multiple clients using significantly less GPU memory compared to duplicating the base model for each client.

Second, GPU resources in split learning are significantly under-utilized, as the computations are constantly interrupted by communications, yet the GPU memory is still preserved. This severely limits the system's scalability to more clients. To address this problem, *Menos* introduces an *on-demand memory allocation mechanism*, which allocates the minimum amount of required GPU memory to clients only when computation is ready. When a client's turn for computation arrives, *Menos* dynamically allocates the necessary GPU memory, performs the forward and backward passes, and immediately release the memory upon completion. During waiting times, GPU memory will be released to allow other clients' tasks to be scheduled. This temporal sharing of GPU memory improves utilization and allows more clients to be served.

Finally, we have implemented *Menos* and conducted an extensive array of experiments to evaluate its performance in a real-world environment. Our results show that, using our base model sharing mechanism, *Menos* substantially reduces GPU memory consumption by up to 72% as the number of clients increases. At the same time, the on-demand memory allocation and scheduling in *Menos* achieves a highly efficient temporal sharing of GPU resources, allowing multiple clients conducting split fine-tuning with limited server GPU memory, while the time overhead is negligible. Our extended experiments in multi-GPU environments and with CPU clients further demonstrate that *Menos* can be widely adopted across different hardware settings.

2 Adapter-Based Split Fine-Tuning

2.1 Adapter-Based Model Fine-Tuning

Due to the enormous number of parameters in large pre-trained models, fine-tuning the entire model is often too computationally expensive for most users [14]. Instead, the prevailing techniques typically rely on adapter-based fine-tuning [10], which means adding a small "adapter" module to the pre-trained model and only updating the adapter parameters, while keeping the original pre-trained model or the base model fixed.

As examples, LoRA (Low Rank Adaptation) [9] injects low-rank parameterized matrices to target model operations, and adapts to the fine-tuning data by optimizing these low-rank matrices without modifying the original model parameters. Prefix-tuning [16] concatenates adapter layers to the beginning of the base model, and other customized adapter modules [8, 17] are also presented in different forms. With these techniques, the adapter modules learn task-specific representations and patterns, avoiding the cost of updating all the base model's parameters.

Formally, assume we have a pre-trained model with parameters θ. We add a small adapter module with parameters ϕ that gets inserted into the pre-trained model such that the model becomes $f(x;\theta,\phi)$, where f is the function represented by the model, and x is the input. With a loss function $L(\theta,\phi)$ defined as

$$L(\theta,\phi) = \text{Loss}\left(f(x;\theta,\phi), y\right)$$

where y is the target output or label. In this way, the adapter-based model fine-tuning can be formulated as:

$$\phi^* = \text{argmin}_\phi L(\theta, \phi). \tag{1}$$

So only the adapter parameters ϕ are optimized and the base model parameters θ remain untouched.

2.2 Split Fine-Tuning

Split learning [7], as the term suggests, splits the model training process between the server and clients. As presented in Fig. 1, a model is topologically partitioned into three sections: the input and output sections f_i and f_o, including the input/output layers and some nearby layers, reside on the client-side so that the fine-tuning dataset can be processed locally. The main body of the model f_s will be hosted on the server-side.

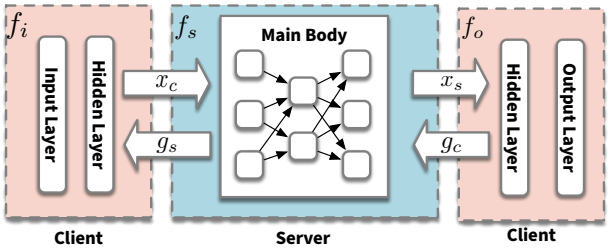


Figure 1. The split fine-tuning process.

Clients start fine-tuning by feeding input data x into the input section f_i to obtain intermediate activations $x_c = f_i(x; \theta_i, \phi_i)$ and sending them to the server. The server continues the forward computation via $x_s = f_s(x_c; \theta_s, \phi_s)$ and return x_s to client. The client then computes the output $f_o(x_s; \theta_o, \phi_o)$ using its output section f_o and calculates the loss to begin back-propagation.

When the gradients $g_c = \frac{\partial L}{\partial x_s}$ from the loss reach the cut point between f_s and f_o, they will be sent to the server to continue gradient propagation through f_s, yielding the gradients g_s with respect to x_c. The server returns g_s to the client to complete gradient computation across all three model sections. With the full set of gradients, the adapter parameters ϕ_i, ϕ_s, and ϕ_o can now be updated with their respective gradients using the selected optimization algorithm.

From the perspective of the server, a complete fine-tuning cycle can be summarized in the following steps:

- **Step 1.** The server waits to receive intermediate activations x_c from the client.
- **Step 2.** The server performs forward computation to obtain x_s, and returns it to the client.
- **Step 3.** The server waits to receive gradients g_c from the client.
- **Step 4.** The server executes back-propagation to get g_s, and returns it to the client.

The four steps above comprise the core fine-tuning loop between the clients and the server in split fine-tuning. This process iterates, with the client and server exchanging activations and gradients, until a predetermined exit criterion is met — such as the model reaching a target performance, or completing a maximum number of iterations.

2.3 GPU Memory Footprint Analysis

During the runtime of a model fine-tuning process, the GPU memory footprint mainly comes from the following components:

- **Model parameters** ($\mathbb{M}$)**.** The original model parameters need to be stored in GPU memory so they can be efficiently accessed and used during fine-tuning. Since modern large pre-trained models can have billions of parameters, this typically consumes the most GPU memory.
- **Adapter parameters** ($\mathbb{A}$)**.** The additional adapter parameters used for model fine-tuning also reside in GPU memory. With existing parameter-efficient adapter techniques, we typically have $\mathbb{A} \ll \mathbb{M}$.
- **Optimizer states** ($\mathbb{O}$)**.** Most optimization algorithms like Adam and SGD maintain momentum buffers and other state variables for each *trainable* parameter. As only the adapter parameters are trained during fine-tuning, the optimizer states are on the same order of magnitude as $\mathbb{A}$, and much smaller than $\mathbb{M}$.
- **Intermediate results** ($\mathbb{I}$)**.** Gradient back-propagation requires access to activations from the forward pass, so these are cached in the GPU memory between the forward and backward passes. Larger batch sizes result in higher activation memory.

In the context of split fine-tuning, the vanilla split learning approaches need to allocate GPU memory for each client, which makes the GPU memory consumption linearly grows with the number of clients, *i.e.*, the GPU memory required for N clients can be estimated by

$$(\mathbb{M} + \mathbb{A} + \mathbb{O} + \mathbb{I}) \times N. \tag{2}$$

To understand how much GPU memory is needed in practice, we conducted a measurement study on split fine-tuning using the Llama 2-7B model [32] with the LoRA technique[1]. In this setup, the server offloads the transformer blocks, leaving the remaining model on the client. With a batch size of 4, the server requires approximately 28.7 GB of GPU memory to support fine-tuning. This consists of 24 GB for the base Llama model parameters ($\mathbb{M}$), 246 MB for the adapter parameters and optimizer states ($\mathbb{A} + \mathbb{O}$), and 4 GB for intermediate results ($\mathbb{I}$).

The substantial GPU memory requirement for split fine-tuning large pre-trained models poses significant challenges when scaling to multiple clients. For example, with the Llama

[1]The evaluation section will provide full details on the model architecture, dataset, and fine-tuning configurations.

2-7B model, most high-end server GPUs like the NVIDIA V100 (32 GB) or A100 (40 GB) can only support split fine-tuning for a single client at a time, which significantly limits the scalability of such split fine-tuning systems. To address this problem, the unique features of model fine-tuning and split learning jointly motivate our design of *Menos* to optimize the GPU memory utilization in split fine-tuning scenarios.

3 Framework Design

Based on the measurement results, we can see that the majority of GPU memory during large model fine-tuning is occupied by the base model parameters and intermediate results, while the adapter and optimizer make up a much smaller fraction. Therefore, the high-level design philosophy of *Menos* aims to optimize the GPU memory consumption by efficiently memory sharing.

3.1 Base Model Sharing Mechanism

As we mentioned above, adapter-based approaches are the most common way to fine-tune large models. Their key feature is that the base model parameters remain fixed, while only the adapter parameters are optimized during fine-tuning. In other words, the base model parameters are read-only throughout the process. This naturally motivates sharing the base model across clients to avoid duplication in GPU memory.

Due to the privacy-efficiency trade-off, clients may choose to cut the model at different layers [39]. Clients concerned more about privacy cut the model at deeper layers, exposing less information to the server. Clients focused on efficiency cut earlier to utilize more server resources. Also, clients may choose different fine-tuning methods like LoRA or prefix tuning based on their needs. These methods insert adapter parameters into the model and modify the computation graph to redirect activations through the adapters.

Both model layer cutting and fine-tuning methods require exclusive modifications to the model structure. Once the structure is modified by one client, it cannot be directly reused by other clients. To safely share the base model among different clients, *Menos* separates the model parameters from the model structure. In other words, there are multiple client-specific model structures that can be modified, while the underlying parameters remain in a single shared copy.

Loading a pre-trained model requires two steps: 1) constructing the model according to the model definition, and 2) reading pre-trained model parameters from files and associating them with the model. *Menos* intercepts this process by skipping the process of reading. As presented in Fig. 2, only one copy of the base model (layers in solid boxes) are preloaded into the GPU memory in advance. Whenever new clients arrive, *Menos* creates different model instances (layers in dashed boxes) for them individually. Yet the parameters

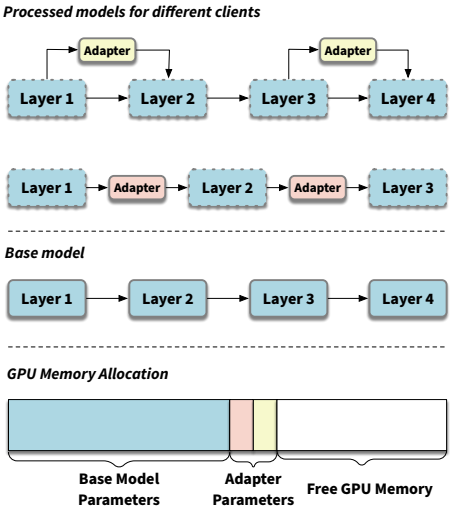


Figure 2. Base model sharing mechanism.

of these models point to the same area in the GPU memory. In this way, the parameters are detached from the model structure, allowing those model instances to be customized for each client (adapters in different colors and structures).

Notice that the GPU memory illustrated in Fig. 2 is an abstraction of all available GPUs, which means *Menos* is compatible with multi-GPU and multi-server environments. For instance, when dealing with multiple GPUs on a single server and an LLM too large to fit into any single GPU, we can manually assign different layers across multiple GPUs while loading the model, which is a fundamental feature provided by most deep learning frameworks. Likewise, for GPUs distributed across multiple servers, we can employ the PyTorch Distributed library to allocate workload among servers and manage internal communications.

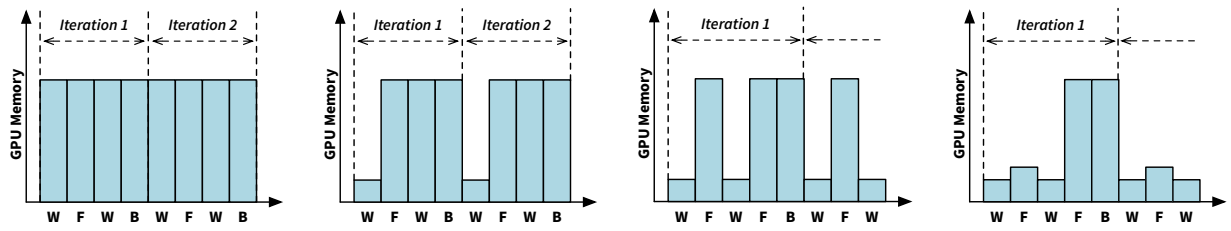
Compared with duplicating the base model for all clients, *Menos* can significantly save more GPU memory for computation purpose. With the base model sharing mechanism, we reduce the GPU memory estimation for N clients from Eq. (2) to

$$\mathbb{M} + (\mathbb{A} + \mathbb{O} + \mathbb{I}) \times N,$$

which takes the most memory consuming part out of the scaling factor. By eliminating the need to duplicate the base model for each client, *Menos* provides a more efficient utilization of GPU resources.

3.2 On-Demand Memory Allocation

In traditional on-device model training, the GPU memory for intermediate results is typically preserved throughout the entire training process. This is efficient because the backward pass immediately follows the forward pass, allowing those cached intermediate results to be immediately accessed for gradient calculations. Preserving this GPU memory ensures



(a) Vanilla (b) Release GPU memory after backward (c) Release GPU memory during waiting time (d) Forward without preparing gradient computation

Figure 3. GPU memory usage patterns in split fine-tuning with different optimization mechanisms. '**W**' indicates waiting for data from clients, '**F**' is forward computation, and '**B**' is backward computation.

the results are available when needed during each backward-forward training step.

In contrast to traditional on-device training, in split fine-tuning the forward and backward passes are interleaved with communication between the server and client. As shown in Fig. 3(a), vanilla split fine-tuning preserves GPU memory for intermediate results even when the server is blocked waiting for information from the client. In this scenario, preserving GPU memory leads to a huge waste, as it remains allocated but underutilized during the waiting time.

To address this inefficiency and optimize resource utilization, *Menos* adopts an *on-demand memory allocation* mechanism. The key idea behind this approach is to dynamically allocate GPU memory for intermediate results only when they are needed during forward and backward passes. When the server becomes blocked waiting for client communication, any unnecessary memory allocation will be freed up to serve other clients. This temporal sharing strategy improves overall GPU utilization compared to the vanilla split learning that preserves all intermediate memory throughout the process.

Our optimization starts by releasing GPU memory after the backward computation is complete. Since the intermediate results for the current optimization round are no longer needed for the subsequent steps, they can be safely freed without modifying the split fine-tuning workflow. As presented in Fig. 3(b), this optimization reduces GPU memory usage during the period when the server is waiting for client activations x_c of next iteration. Note that the base model parameters are shared, and the GPU memory for this client only contains the adapter and optimizer, which require a much smaller amount of GPU memory than the intermediate results.

However, such a short reduction may not offer sufficient opportunity to schedule another client's computation within that time frame. To further optimize the memory efficiency, we can also release the GPU memory while the server waits for gradients g_c, as shown in Fig. 3(c). But as intermediate results needed to compute gradients are discarded, the server must redo the forward pass when g_c arrives for back-propagation. Compared with storing the entire set of intermediate results in GPU memory, we just need to cache the forward activations for the re-forward computation, which is negligible. This process inevitably increases computation, but it allows for more efficient memory utilization during the waiting period, and we will show in the evaluation that the benefit of doing so significantly outweighs the extra computation overhead.

Finally, since activations are recomputed, caching them during the initial forward pass becomes unnecessary. Thus, the server can conduct the first forward pass in a *non-gradient* environment, only generating the activations x_s needed by clients without preparing for gradient back-propagation, which requires much less GPU memory than the typical forward computation. In this way, the GPU memory usage pattern becomes as depicted in Fig. 3(d).

As a result of these optimizations, *Menos* only allocates the minimum required memory to clients for computation. As shown in Fig. 3(d), GPU memory usage stays low for most of the split fine-tuning iteration, leaving ample room to schedule other clients' memory-intensive operations — the second forward and backward passes — during this time. And compared with Fig. 3(a), the peak memory usage only happens in a very short period, which achieves an efficient temporal sharing of GPU memory.

In summary, the computation pattern of the split learning paradigm inherently leaves huge idle time on the GPU, which traditional split learning fails to utilize. *Menos* takes the advantage of this idle GPU time to share the resource across multiple clients. Combining with base model sharing mechanism for adapter-based model fine-tuning, *Menos* reduces the minimum GPU memory requirement for N clients to

$$M + (\mathbb{A} + \mathbb{O}) \times N + I, \tag{3}$$

where we have already shown that $(\mathbb{A} + \mathbb{O})$ is much smaller than M and I thereby increases much slower when the client number N scales up.

3.3 Framework Architecture

Combining the base model sharing mechanism and on-demand memory allocation, we are now ready to present the framework architecture of *Menos* in Fig. 4. *Menos*' workflow begins with the client sending the fine-tuning configurations to the server for profiling. *Menos* enforces a strict on-demand memory allocation policy, therefore it is crucial for the server to know the exact amount of GPU memory required for forward and backward computation to avoid out-of-memory errors.

Given that base model parameters are preloaded and shared, there are two important factors that determine runtime GPU memory demands:

1. Adapter settings, such as LoRA ranks and target layers, which affect the size of the model adapter (i.e., $\mathbb{A}$ in Eq. (3)).
2. Fine-tuning settings, such as optimizer hyperparameters, batch size, and maximum sequence length, which affect the memory consumption of the optimizer states and intermediate results (i.e., $\mathbb{O}$ and $\mathbb{I}$ in Eq. (3)).

When a new client connects to the server, it will first report its fine-tuning configurations. The server then initializes the adapter and optimizer for the client. To accurately profile the runtime GPU memory requirements, the server generates random input sequences based on the reported configurations (e.g., sequence length and batch size). These sequences are passed through forward and backward computations to measure the GPU memory demands for these operations.

Since fine-tuning configurations typically are constant throughout the fine-tuning process, the GPU memory demand remains static. As a result, the *Menos* server only needs to profile the client once, then it can use these measurements as an accurate reference during runtime. It's worth noting that during profiling, the *Menos* server only feeds the model with random input sequences with no need to know the information of the model to be fine-tuned, making this process generic and applicable to any combination of model and adapter.

After profiling clients and obtaining knowledge of the GPU memory demands for each client, the server will start to serve the computation tasks from all clients. Clients fine-tune models with their respective local datasets, and send forward and backward requests to the server. These requests arrive at the server accompanied by different data inputs to be processed, specifically:

- Intermediate activations for forward computations (colored purple in Fig. 4).
- Gradients for backward computations (colored red in Fig. 4).

These requests will be placed in a queue maintained by the task scheduler, whose job is to decide which task shall be scheduled based on the GPU memory demands obtained from the profiling phase, as well as the real-time available GPU memory by actively monitoring the GPU status.

As shown in the GPU activities in Fig. 4, all computation tasks conclude by releasing the occupied GPU memory of the intermediate results. The scheduler captures these updates, and if the remaining GPU memory is sufficient, it schedules one or more computation tasks to run on the GPU. The design details of the scheduler will be elaborated in the next section, and we present a brief working logic of *Menos*' serving process in Algorithm 1.

Algorithm 1 *Menos*' serving process.

```
Initialize model f(θ) from shared parameters θ
Crop the layers and inject adapter parameters f_s(θ_s, φ_s)
while fine-tuning is not done do
    Wait for client's activations x_c
    if scheduled then
        Compute x_s = f_s(x_c; θ_s, φ_s) without caching activations
        Release GPU memory
    Return x_s and wait for client's gradients g_c
    if scheduled then
        Forward x_s = f_s(x_c; θ_s, φ_s) with gradient preparation
        Backward from g_c to obtain g_s
        Optimize the server side adapter φ_s
        Release GPU memory
        Return g_s to client
```

4 Task Scheduling

4.1 Overview

Menos dynamically requests GPU memory on demand instead of preserving GPU memory for forward and backward computations, therefore we need a scheduler to coordinate the computation tasks and available GPU memory such that the memory can be efficiently utilized and reused among clients and computation tasks. Unlike traditional GPU scheduling problems that operate at job level where job lengths can be hours [6, 18], *Menos* schedules at the operation level, where operations like forward and backward computations finish very quickly. These operations arrive at a high frequency, typically requiring just few seconds per round.

The high frequency and short duration of these operations pose unique challenges that *Menos* scheduler has to solve. It must be able to assess the memory requirements of incoming operations, allocate available memory efficiently, and coordinate the execution of operations to avoid conflicts and optimize throughput. This requires a sophisticated scheduling algorithm that can make decisions in real-time based on the current state of the system and the characteristics of the pending operations.

We can summarize the key principles that guide the *Menos* scheduler as follows:

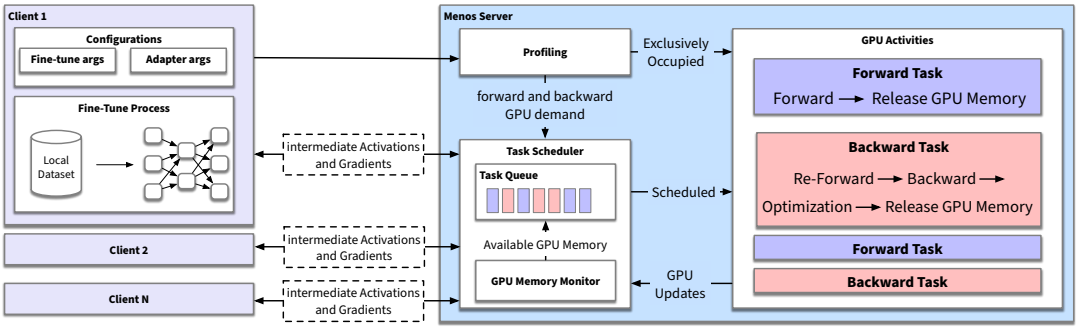


Figure 4. Framework architecture of *Menos*.

1. Avoid out-of-memory errors by tracking memory usage and allocating memory only when it is available.
2. Maximize the utilization of GPU memory by releasing and reusing memory blocks whenever possible.
3. Make quick scheduling decisions to keep up with the requests and prevent the system from being blocked by the scheduler.

With these principles in mind, we next introduce the implementation details of the *Menos* scheduler.

4.2 Implementation Details

Algorithm 2 *Menos* scheduler.

```
1:  Gather the available GPU memory M_avail
2:  Measure the memory demands M_f and M_b for each client
3:  Initialize waiting list W = []
4:  Initialize allocation dictionary A = {}
5:  while True do
6:      Block and wait for trigger event.
7:      if triggered by incoming data from client i then
8:          Add S_i to waiting list W
9:          Call schedule procedure
10:     if triggered by task completion of client i then
11:         Reclaim available memory M_avail = M_avail + A[i]
12:         Clear the allocation entry for client i A[i] = 0
13:         Call schedule procedure
14: procedure SCHEDULE
15:     if waiting list W is not empty then
16:         Select the first request in waiting list s = W[0]
17:         m = M_f[i] if s requests forward else M_b[i]
18:         if GPU memory is sufficient M_avail ≥ m then
19:             Send scheduling signal to s
20:             Reduce available memory M_avail = M_avail − m
21:             Set the allocated memory A[s] = m
22:             Remove s from waiting list
23:             for remaining clients r in waiting list W do
24:                 Schedule r if remaining memory is sufficient
```

Assume there are N clients conducting split fine-tuning with the server, whose corresponding serving processes are identified by $S_1, S_2, \ldots, S_N$. To properly schedule operations, we first need to measure the GPU memory consumption of their forward and backward computations. Given a fixed model, batch size, and sequence length, the GPU memory required for the forward pass $M_f[i]$ and backward pass $M_b[i]$ of process S_i is constant. To track the current GPU memory usage, *Menos* maintains the available memory M_{avail} and an allocation dictionary A where $A[i]$ is the memory allocated to serving process S_i.

The scheduling logic is presented in Algorithm 2. The scheduler is event-driven, triggered by the following two events:

1. Activations or gradients arrive from client i, indicating the input data of forward or backward computation is ready. In this case, the serving process S_i is added to the waiting list (Lines 7-9).
2. An ongoing computation finishes and frees up GPU memory. The scheduler will reclaim the allocated memory (Lines 10-13).

These two events represent the behaviors of requesting and releasing the GPU memory, indicating a potential waiting task can be scheduled within the remaining GPU resources, therefore trigger the scheduling logic to pick a pending task for execution.

To achieve the three principles we mentioned above, *Menos* borrows the scheduling logic from [23] which combines FCFS (first-come-first-serve) and backfilling strategy, and adapts it to fit the application scenario of *Menos*. If the waiting list is not empty, *Menos* checks the first request in the waiting list and the available GPU memory (Lines 15-18). If the available GPU memory is not enough to fulfill the first request in the line, *Menos* quits the current scheduling cycle and waits for more available memory. This FCFS behavior guarantees that memory-intensive operations (e.g. backwards with large batch sizes) will not be starved.

However, if the first request can be fulfilled with the available memory, *Menos* sends the scheduling signal to the corresponding serving process and adjusts the values of scheduler variables (Lines 19-22). It then performs backfilling by scanning the remaining requests in the waiting list to see if the remaining GPU memory can support more clients. If so, those requests are directly scheduled regardless of their positions on the waiting list (Lines 23-24) to fully utilize the available resources.

As we have shown in Fig. 3(d), the re-forward and backward operations can take significantly larger GPU memory than the initial forward operations. Therefore, in a scheduling queue where backward requests are mixed with less demanding forward requests, the FCFS logic prevents long-waiting backward requests from being consistently bypassed by newer, smaller forward requests.

On the other hand, once heavy backward operations are scheduled, the remaining GPU memory may not be enough to support another backward computation, but able to allow forward computations since they take less GPU memory. In this case, the backfilling mechanism takes advantage of any remaining GPU memory to schedule additional requests, even if they arrive later, thereby improving overall system throughput. As a result, this combination of FCFS and backfilling ensures that no clients are starved while still optimizing GPU memory utilization.

Note that while this scheduling method may not result in the optimal GPU memory utilization, it achieved a good balance between fairness, efficiency, and speed. In practice, we implemented *Menos* framework [2] and scheduler using Python and PyTorch [25], where the scheduler takes less than 0.1 milliseconds to make a decision, which is critical for a real-time system where fast scheduling decisions are essential to keep up with the high frequency of incoming requests.

5 Evaluation

5.1 Evaluation Settings

Experimental environment. To evaluate our system under real-world conditions, we conduct experiments in a geo-distributed client-server environment connected over the Internet. The clients are situated in Toronto on a machine with 36 GB of RAM, an Intel (R) Xeon (R) Silver 4210R CPU, and one NVIDIA RTX A4500 GPU with 20 GB of GPU memory. On the other side, the server is hosted on a GPU compute node located in Vancouver at the Cedar cluster[3], configured with 128 GB of RAM, 8 virtual CPU cores, and an NVIDIA V100 Volta GPU with 32 GB of GPU memory.

[2]Code will be released soon at https://github.com/iQua/split-fine-tuning
[3]https://docs.alliancecan.ca/wiki/Cedar

Models and datasets. To thoroughly evaluate our system across models with different computational needs, we select two language models for evaluation to represent different scales of model size and computational requirements. The first is OPT [38], the version with 1.3 billion parameters, which represents smaller models that do not consume extensive GPU memory during fine-tuning. The second is Llama 2 [32], comprising 7 billion parameters and representing cutting-edge, gigantic models that demand substantial resources to fine-tune. The main evaluation results are obtained with wikitext-2-raw-v1 dataset [21], but we also demonstrate the convergence results of *Menos* with Tinyshakespeare [13].

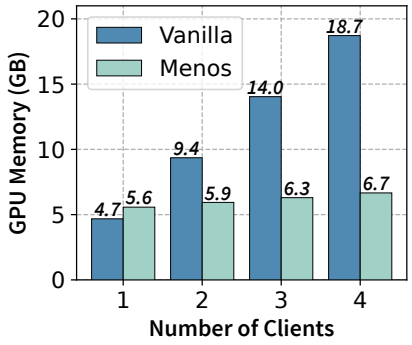
Fine-tuning configurations. We use the most prevailing LoRA technique as the fine-tuning method in our experiments. To maintain the consistency of experimental results, all the clients share the same LoRA configurations borrowed from [19], where the LoRA parameters will be injected to the query and value projections in the transformer layer with $r = 8$ and $\alpha = 16$. For both OPT and Llama 2, we split the model as presented in Fig. 1. The embedding layer, output layer, and the first transformer layer are computed on the client device, while the remaining transformer layers are located on the server. Batch sizes for OPT and Llama 2 are 16 and 4 respectively. Under these settings, the transmission size of intermediate activations and gradients is 13.1 and 12.5 MB for OPT, 6.4 MB and 6.2 MB for Llama 2.

Comparison. *Menos* aims to optimize GPU memory utilization for split fine-tuning to serve more clients within the same memory capacity. To the best our knowledge, no existing work specifically targets this goal for split learning, and other optimization techniques like quantization and gradient checkpointing are orthogonal to *Menos*, which means *Menos* can adopt these methods for further improvements. Therefore, to enable fair comparison, we implement a commonly used task-level sharing mechanism for vanilla split learning: the server accommodates fine-tuning tasks for all clients if sufficient GPU memory is available. When capacity is exceeded, client tasks are swapped out of GPU memory upon completion of each iteration, allowing new incoming clients to be served.

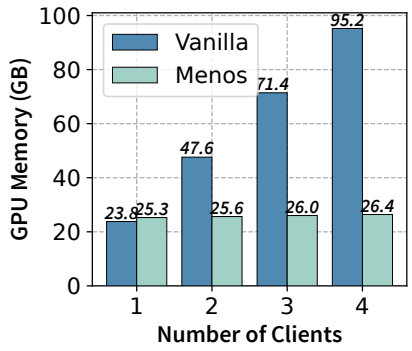
5.2 Evaluation Results

GPU memory reduction. We first present the GPU memory reduction of *Menos*. Since the GPU memory for intermediate results is temporally shared among clients instead of completely removed from GPU memory, for fair comparison, we only compare the GPU memory footprint for components that need to persist in GPU memory during training. This includes the base model parameters, adapter parameters and optimizer states.

As demonstrated in Fig. 5, *Menos* significantly reduces GPU memory consumption compared to vanilla split learning. Without weight sharing, the base model parameters are

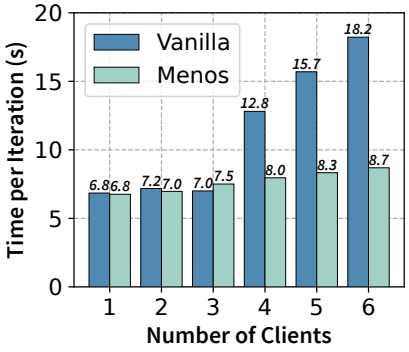


(a) OPT

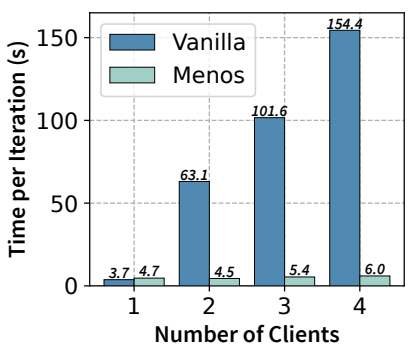


(b) Llama 2

Figure 5. GPU memory consumption for persistent components.

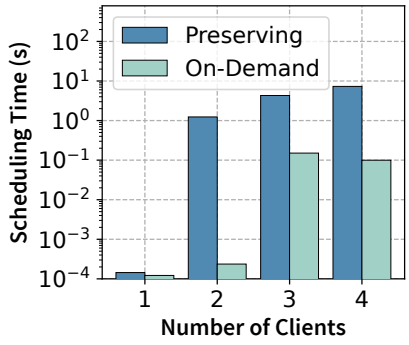


(a) OPT

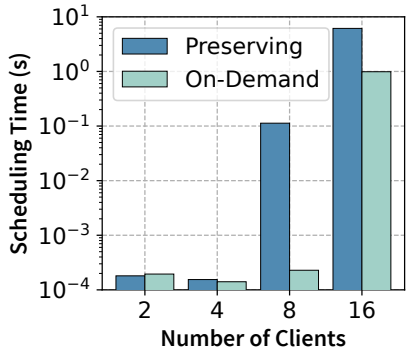


(b) Llama 2

Figure 6. Average time for clients to complete one round of fine-tuning.



(a) OPT



(b) Llama 2

Figure 7. Average schedule time with increasing number of clients.

duplicated for each client in vanilla split learning, leading to a linear GPU memory increase. For example in Fig. 5(a), with 4 clients, the GPU memory for OPT increased from 4.7 GB to 18.7 GB. In contrast, *Menos* only requires 6.7 GB, a 64.1% reduction, by sharing base parameters across clients.

The benefits are even greater for larger models like Llama 2. With 23.8 GB for the full model, vanilla split learning cannot support multiple clients fine-tuning simultaneously with only one NVIDIA V100 GPU unless swapping to main memory or including more GPUs. *Menos* instead needs just 26.4 GB for 4 clients, 72.2% less than duplicating the full model as presented in Fig. 5(b). While quantization techniques like QLoRA [4] and GPTQ [5] are commonly used to reduce the GPU memory demands of large models, these methods could also be applied to the shared model parameters in *Menos* therefore doesn't affect the benefits of *Menos*.

Notice that when there is only a single client, *Menos* uses slightly more GPU memory than vanilla because it requires an extra process to manage the shared base parameters. Vanilla has just one process so requires only one GPU context. But as soon as there are multiple clients, *Menos* provides substantial savings by sharing parameters.

Fine-tuning efficiency. Despite reducing the GPU memory footprint of persistent components, *Menos* temporally shares GPU for the intermediate results through on-demand memory allocation and scheduling during fine-tuning. This naturally raises concerns about the time-space trade-off: does such sharing affect fine-tuning efficiency?

For relatively small models like OPT, as presented in Fig. 6(a), one NVIDIA V100 GPU can support 3 clients simultaneously when each client preserves the required GPU memory. Therefore, with less than 4 clients, the fine-tuning speeds are roughly the same at around 7 seconds per iteration. *Menos* takes about the same time. However, with over 4 clients, the GPU memory is insufficient for vanilla split learning to keep everything on the GPU concurrently. It must offload tasks to main memory to accommodate more clients. This significant task-level swapping slows down the fine-tuning process substantially, increasing the time per iteration to 18.2 seconds with 6 clients. In contrast, scaling the number of clients has a minor impact on *Menos*, requiring only 8.7 seconds with 6 clients fine-tuning simultaneously.

While vanilla split learning performs reasonably well for small models like OPT with few clients, the situation differs drastically for larger models like Llama 2. As mentioned before, one single V100 GPU cannot hold two full copies of the Llama 2 model. Therefore, vanilla split learning must swap tasks starting with just 2 clients. The tremendous time

to transfer the 24 GB model from GPU to main memory increases the per-iteration time from 3.7 seconds to 63.1 seconds. With more clients queuing up, the wait time accumulates, leading to an average of 154.4 seconds per fine-tuning iteration with 4 clients. At 5 clients, even main memory is insufficient, so comparison stops at 4 clients. In contrast, *Menos*' per-iteration time only increases from 4.7 to 6.0 seconds, showing that *Menos* is highly efficient even with a model as large as Llama 2.

Performance analysis. To understand the root causes of *Menos*' performance advantage, we will break down the fine-tuning time into three components: communication time, computation time, and scheduling time.

We first examine communication time. As shown in Table 1, the communication time for a given model remains roughly the same across different numbers of clients. It may take slightly longer as the number of clients increases since clients must share the server's bandwidth, but the impact is negligible. Compared to the overall per-iteration fine-tuning time, we can see that communication is the biggest component for *Menos* and vanilla split learning when GPU memory is sufficient. This is because transmission over Internet is involved.

Table 1. Average communication time (s) per fine-tuning iteration.

Model	Methods	Number of Clients					
		1	2	3	4	5	6
OPT	Vanilla	6.37	6.69	6.84	6.45	6.75	6.43
	Menos	5.93	6.29	7.10	6.73	6.44	6.87
Llama 2	Vanilla	3.23	3.53	3.91	3.66	N/A	N/A
	Menos	3.11	3.47	3.53	3.55	N/A	N/A

Next, we present the computation time in Table 2. Since vanilla split learning preserves all required GPU memory to perform forward and backward computations, computation times are roughly the same for a given model. However, *Menos*' computation time is clearly larger than vanilla split learning. There are two reasons for this. First, *Menos* releases GPU memory after the initial forward computation is complete, then re-conduct the forward pass to obtain the required intermediate results when receiving gradients. Second, *Menos* needs to constantly release GPU memory. As the number of clients increases, GPU memory allocation becomes more fragmented, increasing the time spent on releasing and re-collecting GPU memory. To this end, *Menos* does introduce additional computation overhead compared to vanilla split learning that preserves GPU memory throughout forward and backward passes.

Table 2. Average computation time (s) per fine-tuning iteration.

Model	Methods	Number of Clients					
		1	2	3	4	5	6
OPT	Vanilla	0.41	0.47	0.43	0.52	0.51	0.54
	Menos	0.71	0.75	0.93	1.24	1.46	1.68
Llama 2	Vanilla	0.46	0.52	0.55	0.51	N/A	N/A
	Menos	1.15	1.17	1.53	2.16	N/A	N/A

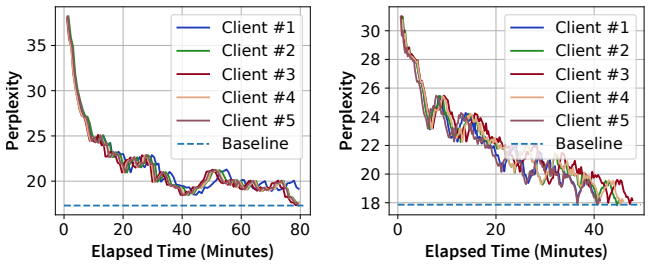
Finally, the scheduling time, defined as the time between when the server receives intermediate activations (or gradients) and starts forward (or backward) computation, is compared in Table 3. For vanilla split learning with OPT, scheduling time is 0 when GPU memory is sufficient. Otherwise, it increases as clients must wait for others to complete their computations and swap out their tasks, resulting in considerable scheduling overhead. In contrast, *Menos* maintains very low scheduling times across all the number of clients. This is because by sharing base model parameters, the saved GPU memory allows clients to be served immediately without waiting, *i.e.*, no scheduling overhead for OPT in our settings.

For Llama 2, the scheduling time for vanilla rapidly increases and accumulates due to the overhead of moving large amounts of data between GPU and main memory. Since the GPU can only support one client, *Menos* clients also need to wait when it is occupied. However, *Menos* does not transfer data between GPU and main memory — clients just wait for previous clients to release GPU memory. Therefore, its scheduling overhead is very small compared to computation and communication times.

There are two scheduling requests for forward and backward computation respectively in *Menos*. We discover that there is almost no waiting time for forward requests even for Llama 2. This is because forward operations require far less GPU memory, and our scheduling algorithm can always select and parallelize them with the backward computations of other clients.

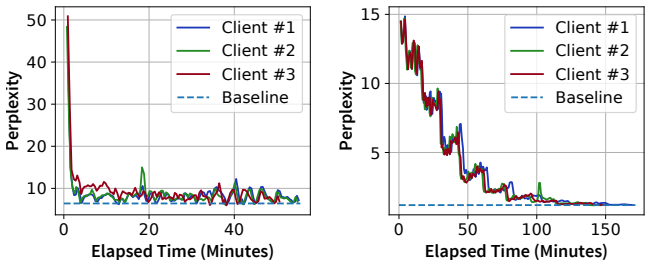
Table 3. Average schedule time (s) per fine-tuning iteration.

Model	Methods	Number of Clients					
		1	2	3	4	5	6
OPT	Vanilla	0	0	0	4.99	7.81	8.18
	Menos	Ranging from 0.000126 ~ 0.000132					
Llama 2	Vanilla	0	39.9	81.6	121.1	N/A	N/A
	Menos	0.0001	0.08	0.25	0.38	N/A	N/A



(a) Wikitext (b) Tiny-Shakespeare

Figure 8. Convergence of OPT 2.



(a) Wikitext (b) Tiny-Shakespeare

Figure 9. Convergence of Llama 2.

On-demand allocation vs. memory preserving policies. Our evaluations so far have validated the effectiveness of *Menos*' design in most cases. However, the on-demand memory allocation policy remains controversial since it inevitably increases the time spent on computation. However, we argue that this overhead is preferable compared with preserving intermediate results during waiting for client-side gradients.

Fig. 7 compares our design with the memory preserving policy. For OPT, the scheduling time is less than 1 millisecond with 2 and 4 clients, and rises to only 0.12 seconds with 8 clients. However, when the client number reaches 16, the scheduling time drastically increases to 6.1 seconds. This is because the time spent waiting for gradients adds up to the queuing delay of those clients waiting to be scheduled. The situation is even worse with Llama 2. The queuing delay occurs when there are only 2 clients and quickly increases to about 10 seconds with just 4 clients.

In contrast, the on-demand memory allocation policy of *Menos* allows queuing clients to avoid suffering from such communication overhead. In practice, the average scheduling times with Llama 2 are only 0.38 seconds with 4 clients, and 1.01 seconds for OPT with 16 clients. Although this is a slight increase, it is still significantly smaller than the communication overhead that the memory preserving policy must endure.

Exploration on multi-GPU environment. In our previous experiments, we limited the client number to a small range as the GPU memory on the client side cannot be shared. Next, we launch clients on CPU devices to increase the scale of clients, and explore the performance of *Menos* in a multi-GPU environment with Llama 2 model.

Our results, as illustrated in Fig. 10, firstly reveal that even without GPU acceleration, the fine-tuning time for 2 clients only slightly increased (from 4.5 to 5.3 seconds) compared to GPU clients shown in Fig. 6(b). This is because there are only minimum computations happened on client side, as most layers are offloaded to *Menos* server, therefore the client side computation power doesn't affect performance too much. This also implies that *Menos* client doesn't have to own a GPU device to fine-tune LLMs.

On the other hand, as the client number increases from 2 to 10, the time for one step of fine-tuning rises from 5.3 to 11.2 seconds when only 1 GPU is available. However, with 4 GPUs, the fine-tuning time is only 6.6 seconds for 10 clients. As mentioned before, more GPUs mean more available GPU memory for *Menos* to schedule. If the GPU memory cannot accommodate all clients simultaneously, such as 1 GPU for 10 clients, GPU memory swapping inevitably slows down the fine-tuning speed. This slowdown goes roughly linearly with the number of clients, and can be avoided by incorporating more GPUs.

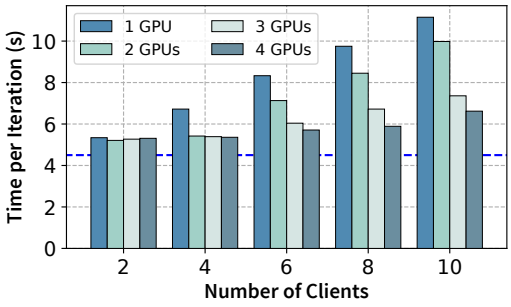


Figure 10. Fine-tuning time with multi-GPU server and scaling clients on CPU device. The blue dashed line represents the baseline time for 2 GPU clients.

Model convergence. Finally, we address the concern about *Menos*'s model convergence. Mathematically, the fine-tuning results of *Menos* are identical to single-device fine-tuning, as it only distributes computation while maintaining the same logical flow. We validate this by examining convergence for multiple clients fine-tuning OPT and Llama 2, including an additional dataset, Tiny-Shakespeare.

Results in Fig. 8 and Fig. 9 show that all clients reached the same final perplexities as local fine-tuning (represented by the lowest dashed blue line), despite taking longer due to cross-internet communication. This demonstrates that

Menos preserves the convergence properties of fine-tuning methods across different models and datasets.

6 Related Work

Model fine-tuning. Model fine-tuning allows transferring knowledge from pre-trained models to downstream tasks without training the model from scratch. With the emerging of large pre-trained models containing billions of parameters, fine-tuning techniques have evolved from retraining the entire model [14, 27] on a new dataset, to freezing most parameters but retraining some layers [37], and finally to adapter-based methods [8–10, 16] which update only small adapter modules while freezing the full model. However, these fine-tuning methods are typically conducted in a centralized environment, requiring users to send local data to centralized servers, or model owners to open-source their models. Both scenarios are undesirable, which is why we advocate for split fine-tuning in this paper.

Split learning. Split learning [7] was a distributed machine learning paradigm that divides training between clients and a server. Existing research on split learning mainly focused on enabling multiple clients to collaboratively train a model from scratch using their distributed datasets [26, 34]. Some recent works [24, 30, 33] combined split learning with federated learning to improve model accuracy and computation/communication efficiency.

Instead of training a model from scratch using datasets from different sources, fine-tuning tasks start with a powerful pre-trained model and aim to adapt it to a local dataset. Therefore, *Menos* only adopts the computation paradigm of split learning, but has a very different objective compared to existing split learning works.

Federated Learning. Split fine-tuning and federated learning [20] are two distinct approaches to distributed machine learning. While federated learning trains models across multiple devices without sharing raw data, the split fine-tuning method proposed in the *Menos* paper has several key differences. First, federated learning shares model updates with server, while *Menos* exchanges intermediate activations and gradients. Second, multiple clients are involved in a federated learning session, whereas split fine-tuning occurs between each client and server independently. Last and most importantly, federated learning requires each client to hold the entire model, which is challenging for LLMs. In contrast, split fine-tuning allows fine-tuning of large LLMs with limited client resources.

GPU memory optimization. With the trend of large models becoming even larger, reducing GPU memory usage during training has become an important research area. Methods like gradient checkpointing [1] can reduce memory needed during back-propagation by only storing a subset of activations from the forward pass. Selective re-computation [15] improved this by storing activations that are space-efficient but expensive to recompute while recomputing activations that are space-intensive but cheap to recompute.

However, for fine-tuning tasks, pre-trained model parameters typically dominate GPU memory consumption. Therefore, quantization has commonly been used to reduce the bit depth of model parameters. For instances, reducing precision from 32-bit floats to 16-bit cuts memory usage in half [22]. [2] and [3] further reduced the parameters and optimizers to 8-bit. Further, QLoRA [4] pushed the limit to 4 bits, by introducing a novel NormalFloat4 data format. Similarly, GPTQ [5] quantized the model to 3 or 4 bits per parameter based on approximate second-order information, yet incurring negligible accuracy degradation compared with the original model. These methods are orthogonal to *Menos*, which implies they can be combined with *Menos* for further improvements.

GPU sharing and scheduling. While the aforementioned techniques sought to optimize GPU memory usage for individual tasks, GPU sharing becomes essential as the number of concurrent tasks increases. [12] and [6] analyzed deep neural network training workloads to schedule and allocate GPUs in multi-tenant clusters. [11] improved upon scheduling algorithms to reduce job completion times. These methods schedule machine learning tasks spanning minutes to hours, with the allocation unit being GPUs rather than GPU memory. In contrast, *Menos* requires more fine-grained GPU sharing and scheduling since it works on an operation-level schedule. KubeShare [36] and KubeKnots [31] provided GPU memory sharing at the container level, but they were still designed for general machine learning tasks, while *Menos* is tailored for the computation and communication patterns of split fine-tuning tasks.

7 Concluding Remarks

Split fine-tuning allows large pre-trained models to be fine-tuned without requiring model users or model owners to share private data or model parameters. However, as the number of clients scales up, the massive model size can overwhelm GPU-equipped servers, driving costs higher for providing such split fine-tuning services. To address this challenge, we present *Menos*, the first memory-efficient framework optimized for split fine-tuning across multiple clients. *Menos* strategically utilizes the unique features of split learning and model fine-tuning to achieve highly efficient resource sharing over limited GPU memory. Our extensive experiments in real-world settings have clearly demonstrated its effectiveness, in that it significantly reduces the GPU memory footprint — yet incurring negligible overhead — as the number of clients increases. As the demand for model fine-tuning increases over time, we expect that *Menos* will substantially reduce operating expenses for providing fine-tuning services in a privacy-preserving fashion with respect to each client's data.

References

[1] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *arXiv preprint arXiv:1604.06174* (2016).

[2] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. GPT3.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=dXiGWqBoxaD

[3] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022. 8-bit Optimizers via Block-wise Quantization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=shpkpVXzo3h

[4] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. QLoRA: Efficient Finetuning of Quantized LLMs. *arXiv preprint arXiv:2305.14314* (2023).

[5] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Quantization for Generative Pre-trained Transformers. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=tcbBPnfwxS

[6] Juncheng Gu, Mosharaf Chowdhury, Kang G Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. 2019. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 485–500.

[7] Otkrist Gupta and Ramesh Raskar. 2018. Distributed Learning of Deep Neural Network over Multiple Agents. *Journal of Network and Computer Applications* 116 (2018), 1–8.

[8] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*. PMLR, 2790–2799.

[9] 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 *International Conference on Learning Representations*.

[10] Zhiqiang Hu, Yihuai Lan, Lei Wang, Wanyu Xu, Ee-Peng Lim, Roy Ka-Wei Lee, Lidong Bing, and Soujanya Poria. 2023. LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models. *arXiv preprint arXiv:2304.01933* (2023).

[11] Changho Hwang, Taehyun Kim, Sunghyun Kim, Jinwoo Shin, and KyoungSoo Park. 2021. Elastic Resource Sharing for Distributed Deep Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 721–739.

[12] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 947–960. https://www.usenix.org/conference/atc19/presentation/jeon

[13] Andrej Karpathy. 2015. char-rnn. https://github.com/karpathy/char-rnn.

[14] Jacob Devlin Ming-Wei Chang Kenton and Lee Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proc. NAACL-HLT*. 4171–4186.

[15] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2023. Reducing Activation Recomputation in Large Transformer Models. *Proceedings of Machine Learning and Systems* 5 (2023).

[16] Xiang Lisa Li and Percy Liang. 2021. Prefix-Tuning: Optimizing Continuous Prompts for Generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*. 4582–4597.

[17] Bingyan Liu, Yifeng Cai, Hongzhe Bi, Ziqi Zhang, Ding Li, Yao Guo, and Xiangqun Chen. 2023. Beyond Fine-Tuning: Efficient and Effective Fed-Tuning for Mobile/Web Users. In *Proceedings of the ACM Web Conference 2023*. 2863–2873.

[18] Nils Lukas, Ahmed Salem, Robert Sim, Shruti Tople, Lukas Wutschitz, and Santiago Zanella-Béguelin. 2023. Analyzing Leakage of Personally Identifiable Information in Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 346–363.

[19] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. 2022. PEFT: State-of-the-art Parameter-Efficient Fine-Tuning methods. https://github.com/huggingface/peft.

[20] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Artificial Intelligence and Statistics*. PMLR, 1273–1282.

[21] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[22] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1gs9JgRZ

[23] Ahuva W. Mu'alem and Dror G. Feitelson. 2001. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems* 12, 6 (2001), 529–543.

[24] Seungeun Oh, Jihong Park, Praneeth Vepakomma, Sihun Baek, Ramesh Raskar, Mehdi Bennis, and Seong-Lyun Kim. 2022. Locfedmix-sl: Localize, Federate, and Mix for Improved Scalability, Convergence, and Latency in Split Learning. In *Proceedings of the ACM Web Conference 2022*. 3347–3357.

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS-W*.

[26] Maarten G Poirot, Praneeth Vepakomma, Ken Chang, Jayashree Kalpathy-Cramer, Rajiv Gupta, and Ramesh Raskar. 2019. Split Learning for Collaborative Deep Learning in Healthcare. *arXiv preprint arXiv:1912.12115* (2019).

[27] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[28] Rohan Taori and Ishaan Gulrajani and Tianyi Zhang and Yann Dubois and Xuechen Li and Carlos Guestrin and Percy Liang and Tatsunori B. Hashimoto . 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

[29] Albert Yu Sun, Eliott Zemour, Arushi Saxena, Udith Vaidyanathan, Eric Lin, Christian Lau, and Vaikkunth Mugunthan. 2023. Does Fine-Tuning GPT-3 with the OpenAI API Leak Personally-Identifiable Information? *arXiv preprint arXiv:2307.16382* (2023).

[30] Chandra Thapa, Pathum Chamikara Mahawaga Arachchige, Seyit Camtepe, and Lichao Sun. 2022. Splitfed: When Federated Learning meets Split Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 8485–8493.

[31] Prashanth Thinakaran, Jashwant Raj Gunasekaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. 2019. Kube-knots: Resource Harvesting through Dynamic Container Orchestration in GPU-Based Datacenters. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–13.

[32] 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).

[33] Valeria Turina, Zongshun Zhang, Flavio Esposito, and Ibrahim Matta. 2021. Federated or Split? a Performance and Privacy Analysis of Hybrid Split and Federated Learning Architectures. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 250–260.

[34] Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. 2018. Split Learning for Health: Distributed Deep Learning without Sharing Raw Patient Data. *arXiv preprint arXiv:1812.00564* (2018).

[35] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. 2022. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research* (2022). https://openreview.net/forum?id=yzkSU5zdwD Survey Certification.

[36] Ting-An Yeh, Hung-Hsin Chen, and Jerry Chou. 2020. KubeShare: A Framework to Manage GPUs as First-Class and Shared Resources in Container Cloud. In *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing* (Stockholm, Sweden) *(HPDC '20)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/3369583.3392679

[37] Elad Ben Zaken, Yoav Goldberg, and Shauli Ravfogel. 2022. BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. 1–9.

[38] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. 2022. OPT: Open Pre-Trained Transformer Language Models. *arXiv preprint arXiv:2205.01068* (2022).

[39] Zongshun Zhang, Andrea Pinto, Valeria Turina, Flavio Esposito, and Ibrahim Matta. 2023. Privacy and Efficiency of Communications in Federated Split Learning. *IEEE Transactions on Big Data* (2023).