COMMUTEPROP: DECOUPLED TRAINING FOR COMMUNICATION BOUND SPLIT LLM FINE-TUNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Split learning has emerged as a promising paradigm for privacy-preserving LLM fine-tuning, yet its practical deployment is severely hindered by the sequential communication-computation bottleneck. In conventional synchronous pipelines, clients remain idle while waiting for server-side gradients, resulting in substantial training inefficiency. We propose CommuteProp, an asynchronous split-learning algorithm that decouples the training process into two concurrent phases: a cross-block forward-backward pass and an in-block weight update. By overlapping computation with communication, CommuteProp reduces the marginal cycle time from a sum of all stage latencies to the dominant computational bottleneck. We provide a rigorous asynchronous error and convergence analysis. Moreover, we derive an NS preconditioner method based on our analysis to further mitigate staleness-induced noise. Comprehensive experiments indicate that our algorithm achieves substantial throughput gains while maintaining accuracy comparable to synchronous methods; furthermore, it functions as a plug-and-play module that not only accommodates but actively enhances existing privacy enhancement methods.

1 INTRODUCTION

The increasing demand for fine-tuning Large Language Models (LLMs)(Vaswani et al., 2017) on proprietary datasets has encountered a fundamental conflict between model utility and data privacy. In highly regulated sectors such as healthcare and finance, raw data is highly regulated from leaving local custody due to stringent privacy protocolsVoigt & Von dem Bussche (2017). While Federated Learning (FL) (Kairouz & McMahan, 2021) offers a decentralized alternative, the colossal memory requirements of LLMs often exceed the hardware limits of local clients, rendering full-model local training infeasible. Consequently, Split Learning (SL) (Gupta & Raskar, 2018) has emerged as a critical paradigm. By partitioning the model across institutional boundaries, SL allows clients to retain sensitive data locally while offloading the memory-intensive computation to a powerful server. Specifically, let the model $\mathcal{F}$ be partitioned into N layers with parameters $\theta_1, \ldots, \theta_N$, let $\mathcal{D}$ be the data distribution with samples $(\boldsymbol{x}, \boldsymbol{y})$, and let $\mathcal{L}_{\text{task}}$ denote the loss function. We have the following mathematical problem:

$$\begin{gathered}\min_{\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_N} \quad \mathbb{E}_{(\boldsymbol{x},\boldsymbol{y})\sim\mathcal{D}}\left[\mathcal{L}_{\text{task}}\left(\mathcal{F}(\boldsymbol{x};\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_N),\ \boldsymbol{y}\right)\right] \\ \Theta_{\text{client}} = \{\boldsymbol{\theta}_i \mid i \in \mathcal{I}_c\}, \Theta_{\text{server}} = \{\boldsymbol{\theta}_j \mid j \in \mathcal{I}_s\}, \quad \mathcal{I}_c \cup \mathcal{I}_s = \{1,\ldots,N\}, \mathcal{I}_c \cap \mathcal{I}_s = \emptyset\end{gathered} \tag{1}$$

Parameters on the client side where the private data is hosted are included in Θ_{client}, while bulk part of parameters include in Θ_{server} are on the server side that has no access to the private data. Although SL offers a promising privacy-preserving paradigm, it is severely bottlenecked by substantial communication overhead. In the conventional synchronous execution, the client remains idle during the server-side computation and the subsequent backpropagation of gradients. Specifically, let t_{client} and t_{server} denote the local processing times for the client and server, respectively, and t_c represent the one-way communication latency. The marginal cycle time for a single batch under this synchronous paradigm is strictly additive $\Delta T_{\text{sync}} = t_{\text{client}} + t_{\text{server}} + 2t_c$. Unlike the centralized settings that t_c can be ignored due to the high-speed interconnects (e.g. NVLink), in SL, t_c indicates that the training process faces sever communication overhead **in every iteration**.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

To address these inefficiencies, we propose **CommuteProp**, an asynchronous SL algorithm designed to overlap computation with communication. We decompose the training process into two distinct phases: *cross-block (Phase1)* and *in-block (Phase2)* forward-backward passes. This decoupling allows the client to utilize its previously idle cycles to execute in-block operations for subsequent micro-batches. Ideally, the client no longer waits for the server's gradient feedback to initiate the next forward pass. Consequently, the steady-state cycle time is theoretically governed by the dominant computational bottleneck $\Delta T_{async} = \max(t_{\text{client}}, t_{\text{server}}, t_c)$. Figure 1 demonstrates the superior pipeline efficiency of our proposed algorithm compared to the synchronous baseline in SL.

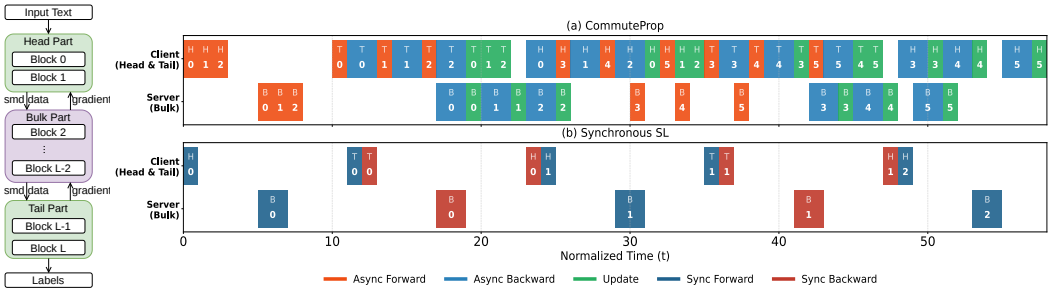


Figure 1: Illustration of pipeline efficiency. Left: USplit(specific scenario of SL) learning overview. Right: Gantt chart comparison between the proposed algorithm and the synchronous baseline in a USplit learning framework. H, B, and T denote the head, bulk, and tail parameters, respectively.

The Gantt chart in Figure 1 indicates that our asynchronous pipeline fully utilizes client resources, whereas the synchronous baseline suffers from substantial idle time due to sequential waiting. While asynchrony inevitably introduces gradient staleness, we provide a rigorous convergence analysis for CommuteProp. Our analysis reveals that the asynchronous approximation error is bounded by a term proportional to the model's cross-block Lipschitz constant — a quantity that prior work Zhang et al. (2024) shows to be typically quite small in LLM. Building on this analysis, we further introduce a preconditioner that actively reduces the asynchronous error.

Comprehensive numerical experiments are conducted to evaluate the empirical performance of CommuteProp across various benchmarking tasks. Our results demonstrate that the proposed algorithm significantly improves training throughput while maintaining competitive model accuracy. Moreover, we provide the ablation study, further shows the impact of gradient staleness on convergence stability, providing insights into the trade-off between asynchrony and optimization trajectory. Furthermore, consistent with recent work Liu et al. (2025) that introduces privacy-preserving methods in SL, our experiment indicates that CommuteProp, as a plug-and-play framework, not only accommodates but actively enhances existing privacy enhancement methods.

We summarize the contribution of this paper as follows: (1) We propose CommuteProp, an asynchronous split-learning algorithm that overlaps computation and communication via a two-phase pipeline; (2) We propose a rigorous asynchronous error and convergence analysis. We also introduce the NS preconditioner method based on our analysis to reduce the asynchronous error; (3) Comprehensive experiments on NLP benchmarks demonstrate substantial throughput gains, competitive accuracy, and plug-and-play compatibility with privacy-enhancing methods.

2 Related Works

SL in LLM Fine-Tuning. Split Learning (SL) Gupta & Raskar (2018); Vepakomma et al. (2018) partitions a model across client and server, allowing the client to offload heavy computation while retaining only a few layers locally. U-Shaped Split Learning (USL) Gupta & Raskar (2018); Lyu et al. (2023) further assigns the head and tail to the client, with the trunk delegated to the server. This architecture is well-suited for autoregressive LLMs. However, recent work Gu et al. (2025); Chen et al. (2024) shows that SL remains vulnerable to data reconstruction attacks, allowing servers to recover portions of the client's private data. This has led to defense mechanisms Liu et al. (2025); Gu et al. (2026) designed to mitigate such risks and strengthen the training process.

Pipeline Parallelism. SL is structurally similar to Pipeline Parallelism (PP). In HPC settings, synchronous PP with micro-batching achieves high throughput and negligible communication overhead Harlap et al. (2018); Qi et al. (2023); Liu et al. (2024). When communication is costly, asynchronous alternatives become preferable, but they suffer from gradient staleness and often require storing historical weights, an impractical storage requirement in SL. Solutions like approximate weight updates Yang et al. (2021) and Nesterov Ajanthan et al. (2025) have been proposed; however, they have only been tested on small models (e.g., nanoGPT Karpathy (2022)). In this work, we adopt synchronous training and the Nesterov-based(without Weight Stashing) method Ajanthan et al. (2025) as baselines, and scale the evaluation to Qwen2.5-1.5B while providing convergence guarantees.

3 METHODOLOGY

3.1 TWO PHASE OPTIMIZATION SCHEME

Without the loss of generality, we decompose the LLM into L blocks. Each of these blocks comprises several transformer blocks, and they are distributed between the client and server nodes. To preserve the residual architecture, we emphasize that individual transformer blocks cannot be split internally. Thus, we can formulate the LLM training into following optimization problem:

$$\min_{\boldsymbol{\theta}_{1,\dots,L}} \quad \mathcal{L}_{\text{task}}(\boldsymbol{h}_L, Y) \quad \text{s.t.} \quad \boldsymbol{h}_{l+1} = \boldsymbol{h}_l + F_l(\boldsymbol{h}_l; \boldsymbol{\theta}_l), \quad \forall l \in \{1, \dots, L-1\} \tag{2}$$

The forward propagation is based on Equation (2). For backpropagation, we relax the strict equality constraints into a quadratic penalty method formulation. Specifically, we denote $\Theta = \{\boldsymbol{\theta}_l\}_{l=1}^{L-1}$ and $H = \{\boldsymbol{h}_l\}_{l=1}^{L}$. We make variables in H free parameters and introduce a punishment term with a coefficient ρ, so the coupled problem is transformed into a globally relaxed objective:

$$\min_{\Theta, H} \quad \mathcal{L}(\Theta, H) = \mathcal{L}_{\text{task}}(\boldsymbol{h}_L, Y) + \frac{\rho}{2} \sum_{l=1}^{L-1} \left\| \boldsymbol{h}_{l+1} - \Big(\boldsymbol{h}_l + F_l(\boldsymbol{h}_l; \boldsymbol{\theta}_l) \Big) \right\|^2 \tag{3}$$

For Equation (3), we employ a two-phase Gaussian-Seidel optimization scheme. We consider the iterative optimization in the time stamp t, and use the superscript $\cdot^{(t)}$ to indicate the value in the current time stamp. Without the loss of generality, we focus on the block $\boldsymbol{h}_{l+1} = \boldsymbol{h}_l + F_l(\boldsymbol{h}_l; \boldsymbol{\theta}_l)$.

Phase 1 (Cross Block). We compute the gradient $g_{l+1} = \nabla_{\boldsymbol{h}_{l+1}} \mathcal{L}(\Theta, H)$. For intermediate blocks, g_{l+1} relies on gradients backpropagated from the subsequent block and is calculated upon receiving them. We then consider a one-step gradient descent with step size $\eta_{\boldsymbol{h}}$: $\boldsymbol{h}_{l+1}^{(t+1)} = \boldsymbol{h}_{l+1}^{(t)} - \eta_{\boldsymbol{h}} g_{l+1}^{(t)}$.

Phase 2 (In Block). We consider optimizing $\boldsymbol{\theta}_l$: $\min_{\boldsymbol{\theta}_l} \left\| \boldsymbol{h}_{l+1}^{(t+1)} - \Big(\boldsymbol{h}_l^{(t)} + F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l) \Big) \right\|^2$. Here, we use $\boldsymbol{h}_{l+1}^{(t+1)}$, $\boldsymbol{h}_l^{(t)}$, because $\boldsymbol{\theta}_l$ is updated after $\boldsymbol{h}_{l+1}$ but before $\boldsymbol{h}_l$. Substituting $\boldsymbol{h}_{l+1}^{(t+1)} = \boldsymbol{h}_{l+1}^{(t)} - \eta_{\boldsymbol{h}} g_{l+1}^{(t)}$ in Phase 1 and forward propagation $\boldsymbol{h}_{l+1}^{(t)} = \boldsymbol{h}_l^{(t)} + F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l^{(t)})$, we obtain:

$$\begin{aligned}
\boldsymbol{\theta}_l^{(t+1)} &= \arg\min_{\boldsymbol{\theta}_l} \left\| \Big(\boldsymbol{h}_{l+1}^{(t)} - \eta_{\boldsymbol{h}} g_{l+1}^{(t)} \Big) - \Big(\boldsymbol{h}_l^{(t)} + F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l) \Big) \right\|^2 \\
&= \arg\min_{\boldsymbol{\theta}_l} \left\| F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l^{(t)}) - F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l) - \eta_{\boldsymbol{h}} g_{l+1}^{(t)} \right\|^2 \\
&= \arg\min_{\boldsymbol{\theta}_l} \frac{1}{2} \left\| F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l^{(t)}) - F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l) \right\|^2 + \eta_{\boldsymbol{h}} \langle F_l(\boldsymbol{h}_l^{(t)}; \boldsymbol{\theta}_l), g_{l+1}^{(t)} \rangle
\end{aligned} \tag{4}$$

$\boldsymbol{h}_{l+1}^{(t+1)}$ and $\boldsymbol{h}_{l+1}^{(t)}$ cancel out in Equation (4), so we only need $\boldsymbol{h}_l^{(t)}$, $\boldsymbol{\theta}_l^{(t)}$ and $g_{l+1}^{(t)}$ to calculate $\boldsymbol{\theta}_l^{(t+1)}$. This allows us to simplify Phase 1 to store merely the per-block activations $\boldsymbol{h}_l^{(t)}$ and gradients $g_{l+1}^{(t)}$ for the subsequent parameter update without the explicit update of $\boldsymbol{h}_{l+1}^{(t+1)}$.

3.2 COMMUTEPROP FRAMEWORK

Under the two-phase Gauss-Seidel optimization scheme discussed above, each block in SL must perform three distinct tasks: in Phase 1, (i) receiving outputs from the preceding block, computing and

transmitting the activations $\boldsymbol{h}_{l+1}$ to the subsequent block during forward propagation; (ii) receiving gradients from the subsequent block, computing and transmitting the gradients g_{l+1} to the preceding block during backpropagation; and in Phase 2, (iii) performing the in-block update of parameters $\boldsymbol{\theta}_l$.

Motivated by this analysis, we propose *CommuteProp* (illustrated in Figure 1), a framework that processes these tasks asynchronously across different data batches. Because the Phase 2 parameter updates rely on the gradients g_{l+1} computed during Phase 1, we establish two separate queues, $\mathcal{Q}_1$ and $\mathcal{Q}_2$, for each block. Phase 1 tasks are managed in $\mathcal{Q}_1$, while the in-block parameter update tasks (Phase 2) are put in $\mathcal{Q}_2$. These queues store only the essential contextual information required for each task, such as $\boldsymbol{h}_l$ and g_{l+1}. To strictly control memory consumption, we cap the maximum length of each queue at $\tau_{\max}$. Furthermore, we exclude the parameter weights $\boldsymbol{\theta}_l$ from the queued task information, as they typically possess a significantly larger memory footprint than either the activations or their gradients.

The exclusion of parameter weights $\boldsymbol{\theta}_l$ from the queued task information introduces challenges when applying Equation (4) to update $\boldsymbol{\theta}_l$. While we can use the delayed gradients $g_{l+1}^{(t-\tau)}$ and activations $\boldsymbol{h}_l^{(t-\tau)}$ retrieved from the queue ($\tau \leq \tau_{\max}$), the absence of the corresponding historical parameters $\boldsymbol{\theta}_l^{(t-\tau)}$ forces the update mechanism to rely on the current parameters $\boldsymbol{\theta}_l^{(t)}$ instead.

For notational simplicity, we define the local loss as $\mathcal{L}_{local}(\boldsymbol{\theta}_l) = \langle F_l(\boldsymbol{h}_l^{(t-\tau)}; \boldsymbol{\theta}_l), g_{l+1}^{(t-\tau)} \rangle$ and assume $F_l(\boldsymbol{h}_l^{(t-\tau)}; \cdot)$ is L_0-Lipschitz continuous: $\|F_l(\boldsymbol{h}_l^{(t-\tau)}; \boldsymbol{\theta}_l) - F_l(\boldsymbol{h}_l^{(t-\tau)}; \boldsymbol{\theta}_l^{(t)})\| \leq L_0 \|\boldsymbol{\theta}_l - \boldsymbol{\theta}_l^{(t)}\|$. We can then reformulate the update rule from Equation (4) as a proximal optimization problem: $\boldsymbol{\theta}_l^{(t+1)} = \arg\min_{\boldsymbol{\theta}_l} \left\{ \eta_{\boldsymbol{h}} \mathcal{L}_{local}(\boldsymbol{\theta}_l) + \frac{L_0^2}{2} \left\| \boldsymbol{\theta}_l - \boldsymbol{\theta}_l^{(t)} \right\|^2 \right\}$. Furthermore, we linearize $\mathcal{L}_{local}(\boldsymbol{\theta}_l)$ at the current parameter $\boldsymbol{\theta}_l^{(t)}$, giving the approximation:

$$\boldsymbol{\theta}_l^{(t+1)} \approx \arg\min_{\boldsymbol{\theta}_l} \left\{ \eta_{\boldsymbol{h}} \left\langle \nabla_{\boldsymbol{\theta}_l} \mathcal{L}_{local}(\boldsymbol{\theta}_l^{(t)}), \boldsymbol{\theta}_l - \boldsymbol{\theta}_l^{(t)} \right\rangle + \frac{L_0^2}{2} \left\| \boldsymbol{\theta}_l - \boldsymbol{\theta}_l^{(t)} \right\|^2 \right\} = \boldsymbol{\theta}_l^{(t)} - \eta \widetilde{g}_l^{(t-\tau)} \tag{5}$$

where $\widetilde{g}_l^{(t-\tau)} = \nabla_{\boldsymbol{\theta}_l} \mathcal{L}_{local}(\boldsymbol{\theta}_l^{(t)}) = \left[\nabla_{\boldsymbol{\theta}_l} F_l(\boldsymbol{h}_l^{(t-\tau)}; \boldsymbol{\theta}_l^{(t)}) \right] g_{l+1}^{(t-\tau)}$ denotes the surrogate gradient, and $\eta = \eta_{\boldsymbol{h}} / L_0^2$ is the effective learning rate which can be tuned in practice.

Notably, we use the current parameter $\boldsymbol{\theta}^{(t)}$ instead of $\boldsymbol{\theta}^{(t-\tau)}$ which is unsaved. A rigorous convergence analysis of this substitution is provided in Section 4. Moreover, although the derivation assumes a first-order update, it naturally extends to popular optimizers such as AdamWLoshchilov & Hutter (2017) and MuonJordan et al. (2024) for LLM fine-tuning. We formalize our two-phase pipeline, CommuteProp, in Algorithm 1.

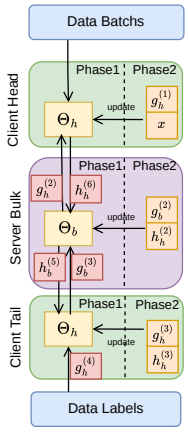


Figure 2: Snapshot of Algorithm 1 on Usplit

Algorithm 1 CommuteProp Node: Event Ingestion

Require: Phase 1 Task Queue $\mathcal{Q}_1$, Phase 2 Task Queue $\mathcal{Q}_2$, Accumulation bound τ

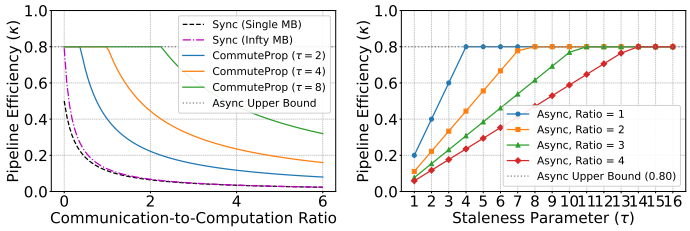
while Training is active **do**
  if Received Backward target $g_{l+1}^{(t)}$ from network **then**
    $\mathcal{Q}_1$.Push(Task_Backward($g_l^{(t)}$)) ▷ Backward pass Θ detached
  else if Received Forward request $\boldsymbol{h}_l^{(t)}$ from network **then**
    $\mathcal{Q}_1$.Push(Task_Forward($\boldsymbol{h}_{l+1}^{(t)}$)) ▷ Forward pass with Θ detached
  end if
  if Receive both $\boldsymbol{h}_{l+1}^{(t)}$ and $g_{l+1}^{(t)}$ **then**
    $\mathcal{Q}_2$.Push(Task_Phase2_Update($\boldsymbol{h}_{l+1}^{(t)}, g_l^{(t)}$) ▷ Update $\boldsymbol{\theta}_l$ based on (5)
  end if
  Task ← **Dispatcher_Schedule**($\mathcal{Q}_1, \mathcal{Q}_2$) ▷ Dispatcher Algorithm 2
  if Task is not NULL **then**
    Execute(Task) ▷ Run the actual computation
  end if
end while

Algorithm 1 outlines the per-node execution logic, while Figure 2 illustrates a snapshot of the overall Usplit workflow at a specific time step. In this figure, the red squares denote the outputs

of cross-block tasks during Phase 1. Conversely, the orange squares represent the inputs derived from these Phase 1 outputs, which subsequently trigger the Phase 2 tasks. Notably, the information communicated between blocks consists strictly of intermediate results ($\boldsymbol{h}_l$ and g_{l+1}) rather than actual model parameters. Because of this, CommuteProp is highly compatible with existing privacy-preserving techniques Liu et al. (2025). This compatibility allows the intermediate data to be safely offloaded to a memory-abundant server, effectively shifting the heavy memory burden away from the client. Finally, given the asynchronous nature of this execution model, efficiently scheduling these distinct tasks to maximize overall pipeline throughput presents a critical challenge, which we address in the subsequent section.

3.3 Dispatcher Schedule and Time Efficiency

In the CommuteProp framework (Algorithm 1), client and server nodes schedule forward activations, backward activation gradients (Phase 1), and weight updates (Phase 2). The scheduling strategy is crucial for computational efficiency, but finding the optimal scheduling strategy is a computationally prohibitive mixed-integer programming (MIP) problem. In this context, based on the Critical Path Method (CPM) (Kelley Jr & Walker, 1959), we introduce a lightweight CPM Heuristic Dispatcher as shown in Algorithm 2. Specifically, Phase 1, which are stored in $\mathcal{Q}_1$, naturally form the critical path because they produce the activations and gradients required for communication. Any delay in them directly stalls downstream stages. Phase 2 weight updates stored in $\mathcal{Q}_2$, by contrast, are strictly local and lie off the critical path. Thus, in Algorithm 2, we assign top priority to Phase 1. To avoid excessive staleness, we also set a maximum latency $\tau_{\max}$ for Phase 2.



(a) (b)

Figure 3: Pipeline efficiency comparison across different system parameters. Figure 3a demonstrate the relationship between pipeline efficiency and communication computation ratio. Figure 3b demonstrate the relationship between τ and pipeline efficiency

Algorithm 2 CPM Heuristic Dispatcher

Require: Phase 1 Queue $\mathcal{Q}_1$, Phase 2 Queue $\mathcal{Q}_2$

```
while Training is active do
    if |Q2| ≥ τ_max then
        Q2.Pop()
    else if |Q1| > 0 then
        Q1.Pop()
    else if |Q2| > 0 then
        Q2.Pop()
    end if
end while
```

In CommuteProp, $t_{l,1}$ and $t_{l,2}$ represent the respective computation durations of Phase 1 and Phase 2 for node l, while t_c denotes the one-way communication time. For classical pipeline synchronous algorithm with micro-batch m, denote the total computation time for node l as $t_{l,s}$. Denote κ_a and κ_s as pipeline efficiency for asynchronous CommuteProp and synchronous algorithms, we have:

$$\kappa_a = \frac{\sum_{l=1}^{L}(t_{l,1}+t_{l,2})}{L\max\left\{\max_l\{t_{l,1}+t_{l,2}\}, \frac{2t_c+\sum_{l=1}^{L} t_{l,1}}{\tau_{\max}}\right\}} \quad \kappa_s(M) = \frac{M\sum_{l=1}^{L} t_{l,s}}{L\left[\sum_{l=1}^{L} t_{l,s}+2t_c+(M-1)(2t_c+\max_l t_{l,s})\right]} \tag{6}$$

In CommuteProp, the denominator's bottleneck term comes either from the heaviest node $\max_l\{t_{l,1}+t_{l,2}\}$ or from the overall communication time $\left(2t_c+\sum_{l=1}^{L} t_{l,1}\right)/\tau_{\max}$. The synchronous denominator, in contrast, incurs a per-node serialisation of $2t_c+\max_l t_{l,s}$ and, additionally, a pipeline-drain cost of $\sum_l t_{l,s}$, which CommuteProp avoids entirely.

Specialising to the Usplit setting ($L=2$), we measure the communication impact via the ratios $r_{\text{async}} = 2t_c/(t_{1,1}+t_{1,2}+t_{2,1}+t_{2,2})$, while for the synchronous counterpart it is $r_{\text{sync}} = 2t_c/(t_{1,s}+t_{2,s})$. Because the split design introduces an extra forward computation, the synchronous pipeline's computational load is heavier; In Figure Figure 3a, we assume in $r_{\text{sync}} \approx 0.75\, r_{\text{async}}$ and server is 1.67 faster than client. Figure 3a shows that CommuteProp yields a significant improvement in pipeline efficiency, while Figure 3b reveals that systems with a higher communication-to-computation ratio require a correspondingly larger $\tau_{\max}$ to achieve optimal pipeline efficiency. In the next section, we theoretically analyze the relationship between the latency threshold $\tau_{\max}$ and the resulting utility.

4 CONVERGENCE ANALYSIS

4.1 PRELIMINARY

We begin by introducing the following assumptions:

Assumption 4.1 (Hierarchical Lipschitz Smoothness)**.** *We assume the loss function $\mathcal{L}$ is twice-differentiable and globally L-Lipschitz smooth, such that its Hessian is bounded by L, i.e., $\|\nabla^2\mathcal{L}(\boldsymbol{\theta})\| \leq L$. Furthermore, we assume a hierarchical smoothness structure: for any individual block l, the block-wise Hessian norm is bounded by L_l, and for any pair of distinct blocks l and j ($l \neq j$), the cross-block Hessian norm is bounded by $H_{l,j} \geq 0$. Formally, we have:*

$$\left\|\nabla^2_{\boldsymbol{\theta}_l\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta})\right\| \leq L_l, \quad \textit{and} \quad \left\|\nabla^2_{\boldsymbol{\theta}_l\boldsymbol{\theta}_j}\mathcal{L}(\boldsymbol{\theta})\right\| \leq H_{l,j} \tag{7}$$

Assumption 4.2 (Bounded Gradients)**.** *We assume that the objective function has bounded gradients. That is, there exists a constant $G > 0$ such that for any parameter state $\boldsymbol{\theta}$, the global gradient norm is bounded by $\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta})\| \leq G$.*

Assumption 4.1 characterizes the hierarchical smoothness properties of the objective function. Crucially, recent empirical and theoretical studies Zhang et al. (2024); Dong et al. (2025) on Transformer architectures demonstrate that the cross-block Hessian norms $H_{l,j}$ are typically orders of magnitude smaller than the block-wise Hessian norms L_l. We empirically verify this property on the evaluated split architectures in Section C.2.

Furthermore, under Assumption 4.2, if we employ a standard first-order optimization method (such as Stochastic Gradient Descent) with a learning rate η, the parameter update rule $\boldsymbol{\theta}^{(t)} = \boldsymbol{\theta}^{(t-1)} - \eta\nabla\mathcal{L}(\boldsymbol{\theta}^{(t-1)})$ naturally implies that the step size for any block $j \in \{1, \ldots, L\}$ is strictly bounded per iteration: $\|\boldsymbol{\theta}_j^{(t)} - \boldsymbol{\theta}_j^{(t-1)}\| \leq \eta G$. Further convergence analysis relies on this bounded update property in our subsequent convergence analysis.

4.2 ASYNCHRONOUS ERROR ANALYSIS

Although CommuteProp updates blocks asynchronously, we analyze its convergence by a virtual serialized sequence in which blocks $1, \ldots, L$ are updated one by one, each using the same delayed gradient as in the real execution. Let $\Theta^{(t)}$ be the parameters after t steps in this virtual sequence ($\Theta^{(0)}$ is the initial point). Then

$$\mathcal{L}(\Theta^{(L)}) - \mathcal{L}(\Theta^{(0)}) = \sum_{t=0}^{L-1}\left(\mathcal{L}(\Theta^{(t+1)}) - \mathcal{L}(\Theta^{(t)})\right) \tag{8}$$

In step t, only one block is updated; the other blocks remain fixed. Writing the loss as a function of $\boldsymbol{\theta}_l$ alone (still denoted $\mathcal{L}(\boldsymbol{\theta}_l)$) and using L_l-smoothness yields

$$\mathcal{L}(\boldsymbol{\theta}_l^{(t+1)}) \leq \mathcal{L}(\boldsymbol{\theta}_l^{(t)}) + \langle\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)}), \boldsymbol{\theta}_l^{(t+1)} - \boldsymbol{\theta}_l^{(t)}\rangle + \frac{L_l}{2}\|\boldsymbol{\theta}_l^{(t+1)} - \boldsymbol{\theta}_l^{(t)}\|^2 \tag{9}$$

Substituting the update equation 5 gives

$$\mathcal{L}(\boldsymbol{\theta}_l^{(t+1)}) \leq \mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 - \left(\frac{\eta}{2} - \frac{L_l\eta^2}{2}\right)\|\widetilde{g}^{(t-\tau_{\max})}\|^2 + \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \widetilde{g}_l^{(t-\tau_{\max})}\|^2 \tag{10}$$

Equation (10) indicates that the per-step descent is bottlenecked by asynchronous error $\|\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \widetilde{g}^{(t-\tau)}\|^2$. Thus, we have the following Lemma:

Lemma 4.3 (Asynchronous Error Bound)**.** *Let $\boldsymbol{\theta}^{(t)} = [\boldsymbol{\theta}_1^{(t)}, \ldots, \boldsymbol{\theta}_L^{(t)}]$ be the state of the model parameters at step t. For a target layer l under a staleness delay of $\tau \leq \tau_{\max}$ steps, the asynchronous proxy gradient $\widetilde{g}_l^{(t-\tau)}$ satisfies the following error bound:*

$$\left\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}^{(t)}) - \widetilde{g}_l^{(t-\tau)}\right\|^2 \leq \left(\eta\tau_{\max}G\sum_{j\neq l}H_{l,j}\right)^2 \tag{11}$$

Lemma 4.3 reveals that the asynchronous error inherently scales with learning rate, the staleness delay τ and the cross-block Hessian norms $H_{l,j}$. As highlighted in Section 4.1, the cross-block curvature $H_{l,j}$ is exceptionally small for $l \neq j$. Consequently, the summation term $\sum_{j\neq l} H_{l,j}$ remains relatively small. Building upon this lemma, we establish the overall convergence guarantee:

Theorem 4.4. *Let $\mathcal{L}^*$ be the global lower bound of the task loss function. Under Assumptions 4.1 and 4.2, if the learning rate satisfies $\eta \leq \frac{1}{L_l}$ (where L_l is the Lipschitz smoothness constant of block l), the CommuteProp algorithm satisfies the following ergodic convergence bound after T iterations:*

$$\frac{1}{T}\sum_{t=0}^{T-1} \|\nabla_{\boldsymbol{\theta}_l} \mathcal{L}(\boldsymbol{\theta}^{(t)})\|^2 \leq \frac{2[\mathcal{L}(\boldsymbol{\theta}_l^{(0)}) - \mathcal{L}^*]}{\eta T} + \eta^2 \tau_{\max}^2 G^2 \left(\sum_{j\neq l} H_{l,j}\right)^2 \tag{12}$$

Theorem 4.4 demonstrates that CommuteProp converges to an ε-approximate stationary point. The approximation error, characterized by $\varepsilon = \mathcal{O}\left(\eta^2 \tau_{\max}^2 G^2 (\sum_{j\neq l} H_{l,j})^2\right)$, acts as a constant noise floor. In practice, as the model approaches the stationary point, the true gradient norm shrinks, meaning the effective local gradient bound $G^{(t)}$ naturally diminishes. In the next section, we will discuss how to further reduce the asynchronous error for preconditioned optimizers.

4.3 Asynchronous Error Reduction Method

The theoretical results in Lemma 4.3 and Theorem 4.4 establish that the convergence of decoupled training is inherently bottlenecked by the accumulation of staleness-induced noise. To further dissect this mechanism, let $\mathbf{M}_j^{(t)}$ denote the adaptive scaling operator of the optimizer at step t (e.g., $\mathbf{M}_j^{(t)} = \mathbf{I}$ for SGD, or $\mathbf{M}_j^{(t)} = \text{diag}(1/\sqrt{\mathbf{v}_t + \varepsilon})$ for AdamW). By unpacking the explicit update step, the asynchronous error bound can be reformulated via the local preconditioned updates:

$$\left\|\nabla_{\boldsymbol{\theta}_l} \mathcal{L}(\boldsymbol{\theta}^{(t)}) - \widetilde{g}_l^{(t-\tau)}\right\|^2 \leq \left(\eta\tau_{\max} \sum_{j\neq l} \left\|\mathbf{H}_{l,j}\mathbf{M}_j^{(t)}\widetilde{g}_j^{(t-\tau)}\right\|\right)^2 \tag{13}$$

Equation (13) directly indicates that the asynchronous error is driven by the interaction between the cross-block Hessian $\mathbf{H}_{l,j}$ and the scaled update direction $\mathbf{M}_j\widetilde{g}_j^{(t-\tau)}$. In standard AdamW, $\mathbf{M}_j$ serves as a purely diagonal approximation of the local inverse covariance. Because it inherently fails to decorrelate the off-diagonal components of the local curvature ($\mathbf{H}_{j,j}$), the update direction $\mathbf{M}_j\widetilde{g}_j$ remains heavily biased toward the local high-curvature directions. When transmitted across blocks, these residual misaligned components can be amplified by the cross-block operator $\mathbf{H}_{l,j}$, exacerbating the asynchronous error accumulation.

Algorithm 3: Kronecker-Block NS Preconditioner

Require: Gradient matrix $U \in \mathbb{R}^{M\times N}$, block size b.

1: Partition U into an $R \times C$ grid of blocks $\{U_{rc}\}_{r,c=1}^{R,C}$.
2: Maintain row covariances $\{V_r\}_{r=1}^{R}$ and column covariances $\{V_c\}_{c=1}^{C}$; row preconditioners $\{P_r\}_{r=1}^{R}$ and column preconditioners $\{P_c\}_{c=1}^{C}$.
3: $V_r \leftarrow \beta V_r + (1-\beta)\sum_c U_{rc}U_{rc}^\top$
4: $V_c \leftarrow \beta V_c + (1-\beta)\sum_r U_{rc}^\top U_{rc}$ ▷ **Low-freq covariance update**
5: $P_r \leftarrow V_r^{-1/4}$ and $P_c \leftarrow V_c^{-1/4}$ via Newton-Schulz iteration. ▷ **Preconditioner refresh**
6: **return** $\tilde{U}$ with blocks $\tilde{U}_{rc} = P_r\, U_{rc}\, P_c$.

To mitigate this, inspired by the Muon optimizer in first-order orthogonalization, we propose to explicitly *whiten* the gradient space prior to applying the diagonal operator $\mathbf{M}_j$. By introducing a preconditioner $\mathcal{P}_j \approx \mathbf{H}_{j,j}^{-1/2}$, we transform the unbounded inter-block covariance into a bounded correlation operator, thereby neutralizing the asynchronous error. To achieve this inverse square root approximation efficiently without heavy eigendecomposition, we introduce the Kronecker-Block Newton-Schulz (NS) Preconditioner. For 2D parameter tensors (e.g., weight matrices), let $\mathbf{U} \in \mathbb{R}^{M\times N}$ denote the reshaped gradient matrix of the block update $\widetilde{g}_l$, as detailed in Algorithm 3.

To ensure efficiency, Algorithm 3 employs a Kronecker-factored structure (inspired by SOAPVyas et al. (2024)) to tractably approximate high-dimensional second-moment statistics without materializing the full Hessian. Furthermore, since the empirical Fisher information evolves slowly during

training Zhao et al. (2024), we implement lazy updates for both covariance aggregation and preconditioner generation. This spatio-temporal design significantly amortizes computational overhead while strictly preserving the spatial decorrelation crucial for robust asynchronous decoupling.

5 EXPERIMENT

Experiment Settings. We use Qwen2.5-1.5B-Instruct as the backbone and partition it with the Usplit architecture. To compare utility after full training without the high wall-clock cost of running every setting on real distributed systems, we use a queue-based pipeline simulator following prior split-learning works Liu et al. (2025); Thapa et al. (2022); Chen et al. (2024). Forward activations are buffered up to depth $\tau_{\max}$ before each gradient update. We consider two settings: *Full Training*, where asynchronous methods complete all epochs, and *Equal Time*, where all methods use the same wall-clock budget ("Time constraint" in Table 1). The simulator assumes an ideal bubble-free schedule for the synchronous baseline, so the reported speedups are conservative. **In section 4, we further supplement the simulation with real client-server experiments under controlled bandwidth and latency, together with PipeDream(Narayanan et al., 2021) comparisons.** Appendix B provides these results, simulation details, datasets, and hyperparameters.

5.1 MODEL UTILITY

Table 1 reports utility across the full-training and Iso-time settings.We use Nesterov Async (Ajanthan et al., 2025), the most recent asynchronous method, as our asynchronous algorithm baseline, as it does not require additional weight stashing similar to our algorithm. AdamW and Muon in Table 1 reports the synchronous result. Rows without "Time constraint" (e.g., AdamW, CommuteProp, Nesterov Async) correspond to full training; those marked "Time constraint" are obtained under the Iso-time budget. Result on other base model are defferd in Appendix C

Table 1: Utility comparison under full-training and Equal time budgets

Algorithm	Classification (Acc)		Generation			
	Emotional	AG-News	SAMSum		SciERC	Spider
	Accuracy	Accuracy	ROUGE	METEOR	F1-Score	EM
Zero-shot	43.33	63.00	23.82	37.82	4.30	36.00
Nesterov Async	76.45	90.56	35.53	38.13	7.73	71.00
AdamW	91.15	94.64	36.61	39.72	72.89	81.40
AdamW (Time)	88.45	94.15	35.42	**40.01**	63.10	80.08
CommuteProp	**90.85**	94.57	36.49	39.45	67.21	**81.60**
CommuteProp (NS)	90.10	**94.61**	**36.56**	39.41	**72.09**	81.00
Muon	93.10	92.97	36.27	39.29	72.12	76.20
Muon (Time)	92.00	**92.05**	35.47	**39.69**	60.68	72.60
CommuteProp (Muon)	**92.60**	92.00	**36.31**	39.18	**65.15**	**73.02**

The result indicates that CommuteProp maintains accuracy comparable to synchronous AdamW across all tasks,while the NS preconditioner further narrows these already small gaps. **Since a large epoch budget is used to evaluate algorithm's utility and stability**, all methods approach nearly convergence under the Iso-time setting, leaving only modest accuracy gaps; the throughput advantage of CommuteProp is more clearly seen in the loss curves (Fig.,4).

5.2 ABLATION STUDY

In this section, we conduct a sensitivity analysis to examine the impact of the staleness parameter $\tau_{\max}$ and the model split architecture(denoted as $[head, bulk, tail]$, representing the number of layers placed in the first client part, the server part, and the final client part) on the performance of CommuteProp. We evaluate the performance **across 36 configurations** on the SciERC dataset.

Figure 4 verified our asynchronous error analysis(Lemma 4.3). Specifically, smaller $\tau_{\max}$ and η lead to a smaller asynchronous error. Moreover, the NS preconditioner markedly reduces the asynchronous error, as its curve lies closer to the synchronous baseline. In addition, our results indicate that the proposed algorithm consistently outperforms the Nadam algorithm.

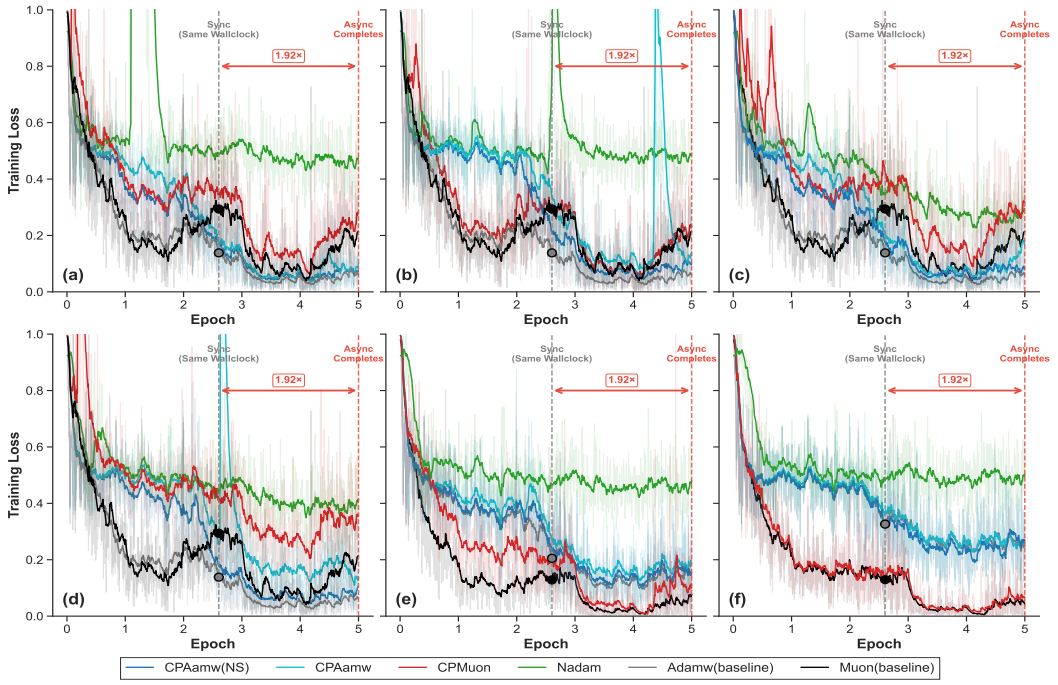


Figure 4: Ablation study on the effects of $\tau_{\max}$, η, and split architecture(CP, short for Commutepro). (**a**–**c**) $\eta = 5 \times 10^{-5}$, architecture $[3, 22, 3]$, with $\tau_{\max} = 16$, 8, and 4, respectively. (**d**) $\eta = 5 \times 10^{-5}$ and $\tau_{\max} = 16$, architecture $[1, 26, 1]$. (**e**–**f**) $\tau_{\max} = 16$, architecture $[3, 22, 3]$, with $\eta = 2 \times 10^{-5}$ and 1×10^{-5}, respectively.

5.3 Privacy Enhancement Integration

Recent work Gu et al. (2025); Chen et al. (2024) shows that SL is vulnerable to gradient matching and Smd data inversion attacks. DualGuard (Liu et al., 2025) was proposed as a defense. Thus, we integrate CommuteProp into DualGuard. Table 8 reports the resulting utility and resilience against various attacks (detailed in Appendix B.7.3). The result indicates that CommuteProp achieves stronger privacy protection than synchronous DualGuard with negligible utility degradation. We attribute this improvement to the implicit regularization induced by stale activations. More experiment result are deferred to Appendix B.7.3

Table 2: Model Utility and Attack Performance.

	Performance		**Attack Method**			
	Rouge	Meteros	SIP	TAG	LAMP	CPTA
w/o defense	86.23	88.48	86.87	65.70	63.13	17.75
Lora DualGuard	85.12	86.92	5.64	5.06	5.81	3.54
Lora CommuteProp	85.18	87.13	5.64	4.34	4.57	3.58

6 Conclusion and Future Work

We proposed CommuteProp, an asynchronous split-learning algorithm that decouples training into cross-block and in-block phases to overlap communication with computation. By exploiting the inherently small cross-block Hessian of LLMs, we theoretically proved that our algorithm is resistant to asynchronous errors, and we accordingly introduced the NS preconditioner to further mitigate residual noise. Experiments show that CommuteProp achieves substantial throughput gains, maintains accuracy, and can integrate with or even enhance existing privacy-preserving methods.

Notably, our theoretical analysis is not confined to Transformers and can be applied to feedforward networks as well. With an appropriate partition, the cross-block Hessian norm could remain bounded, thus carrying over the convergence guarantees. However, identifying an optimal splitting strategy for feedforward architectures remains challenging, because they lack the shortcut mechanism that naturally induces small inter-block interaction. Naive splitting may inflate the cross-block Hessian and break our assumptions. We consider the design of principled splitting rules for general architectures as a meaningful direction for future work.

AI USE STATEMENT

In this work, we used generative AI tools for aiding or polishing writing, implementing methods, summarizing or analyzing existing literature, editing software code. All AI-generated feedback on the scientific content was independently evaluated and verified by the authors, and all final scientific judgments were made by the authors. We have not used generative AI tools for proposing or refining hypotheses, assisting in the writing of proofs, assisting with translation, helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, or supporting qualitative and thematic data analysis. Generating synthetic data sets, cleaning and reformatting datasets, and transcribing recordings of research material are not applicable to this work.

We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REFERENCES

Thalaiyasingam Ajanthan, Sameera Ramasinghe, Yan Zuo, Gil Avraham, and Alexander Long. Nesterov method for asynchronous pipeline parallel optimization. *arXiv preprint arXiv:2505.01099*, 2025.

Guanzhong Chen, Zhenghan Qin, Mingxin Yang, Yajie Zhou, Tao Fan, Tianyu Du, and Zenglin Xu. Unveiling the vulnerability of private fine-tuning in split-based frameworks for large language models: A bidirectionally enhanced attack. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 2904–2918, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Zhaorui Dong, Yushun Zhang, Jianfeng Yao, and Ruoyu Sun. Towards quantifying the hessian structure of neural networks. *arXiv preprint arXiv:2505.02809*, 2025.

Bogdan Gliwa, Iwona Mochol, Maciej Biesek, and Aleksander Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. In *Proceedings of the 2nd Workshop on New Frontiers in Summarization*, pp. 70–79, 2019.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Yujie Gu, Richeng Jin, Xiaoyu Ji, Yier Jin, and Wenyuan Xu. Differentially private and communication efficient large language model split inference via stochastic quantization and soft prompt. *arXiv preprint arXiv:2602.11513*, 2026.

Zixuan Gu, Qiufeng Fan, Long Sun, Yang Liu, and Xiaojun Ye. Vflair-llm: A comprehensive framework and benchmark for split learning of llms. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pp. 5470–5481, 2025.

Otkrist Gupta and Ramesh Raskar. Distributed learning of deep neural network over multiple agents. *Journal of Network and Computer Applications*, 116:1–8, 2018.

Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Keller Jordan, Yuchen Jin, Vlado Boza, Jiacheng You, Franz Cesista, Laker Newhouse, and Jeremy Bernstein. Muon: An optimizer for hidden layers in neural networks, 2024. URL https://kellerjordan.github.io/posts/muon/.

Peter Kairouz and H Brendan McMahan. Advances and open problems in federated learning. *Foundations and trends in machine learning*, 14(1-2):1–210, 2021.

Andrej Karpathy. NanoGPT. https://github.com/karpathy/nanoGPT, 2022.

James E Kelley Jr and Morgan R Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pp. 160–173, 1959.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Zihan Liu, Yizhen Wang, Rui Wang, and Sai Wu. Dualguard: A parameter space transformation approach for bidirectional defense in split-based llm fine-tuning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 17065–17080, 2025.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Yi Luan, Luheng He, Mari Ostendorf, and Hannaneh Hajishirzi. Multi-task identification of entities, relations, and coreference for scientific knowledge graph construction. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 3219–3232, 2018.

Song Lyu, Zheng Lin, Guanqiao Qu, Xianhao Chen, Xiaoxia Huang, and Pan Li. Optimal resource allocation for u-shaped parallel split learning. In *2023 IEEE Globecom Workshops (GC Wkshps)*, pp. 197–202. IEEE, 2023.

Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *International Conference on Machine Learning*, pp. 7937–7947. PMLR, 2021.

Penghui Qi, Xinyi Wan, Guangxing Huang, and Min Lin. Zero bubble pipeline parallelism. *arXiv preprint arXiv:2401.10241*, 2023.

Elvis Saravia, Hsien-Chi Toby Liu, Yen-Hao Huang, Junlin Wu, and Yi-Shin Chen. Carer: Contextualized affect representations for emotion recognition. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 3687–3697, 2018.

Chandra Thapa, Pathum Chamikara Mahawaga Arachchige, Seyit Camtepe, and Lichao Sun. Splitfed: When federated learning meets split learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 36, pp. 8485–8493, 2022.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Praneeth Vepakomma, Otkrist Gupta, Tristan Swedish, and Ramesh Raskar. Split learning for health: Distributed deep learning without sharing raw patient data. *arXiv preprint arXiv:1812.00564*, 2018.

Paul Voigt and Axel Von dem Bussche. The eu general data protection regulation (gdpr). *A practical guide, 1st ed., Cham: Springer International Publishing*, 10(3152676):10–5555, 2017.

Nikhil Vyas, Depen Morwani, Rosie Zhao, Mujin Kwun, Itai Shapira, David Brandfonbrener, Lucas Janson, and Sham Kakade. Soap: Improving and stabilizing shampoo using adam. *arXiv preprint arXiv:2409.11321*, 2024.

Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *Proceedings of Machine Learning and Systems*, 3:269–296, 2021.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3911–3921, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1425. URL https://aclanthology.org/D18-1425.

Xiang Zhang, Junbo Zhao, and Yann LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.

Yushun Zhang, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. Why transformers need adam: A hessian perspective. *Advances in neural information processing systems*, 37:131786–131823, 2024.

Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. Galore: Memory-efficient llm training by gradient low-rank projection. *arXiv preprint arXiv:2403.03507*, 2024.

A Convergence Analysis

To formally bound the asynchronous error caused by the delayed gradients in Phase 2, we first introduce two standard assumptions commonly used in the convergence analysis of non-convex optimization.

Proof of Lemma 4.3. Recall from Section 3.2 that the surrogate gradient used in Phase 2 is defined as

$$\widetilde{g}_l^{(t-\tau)} = \nabla_{\boldsymbol{\theta}_l} L_{\text{local}}\big(\boldsymbol{\theta}_l^{(t)}; \boldsymbol{h}_l^{(t-\tau)}, g_{l+1}^{(t-\tau)}\big),$$

where the activations $\boldsymbol{h}_l^{(t-\tau)}$ and its gradients $g_{l+1}^{(t-\tau)}$ are τ steps stale while the local weight $\boldsymbol{\theta}_l^{(t)}$ is fresh. In the synchronous case ($\tau = 0$) the stale signals coincide with the current ones: $\boldsymbol{h}_l^{(t-0)} = \boldsymbol{h}_l^{(t)}$, $g_{l+1}^{(t-0)} = g_{l+1}^{(t)}$. The surrogate gradient therefore reduces to the standard back-propagation gradient obtained by applying the chain rule at the current, globally synchronous state $\boldsymbol{\theta}^{(t)}$:

$$\frac{\partial \mathcal{L}(\boldsymbol{\theta}_l)}{\partial \boldsymbol{\theta}_l} = \frac{\partial \mathcal{L}(\boldsymbol{\theta}_l)}{\partial \boldsymbol{h}_{l+1}^{(t)}} \frac{\partial \boldsymbol{h}_{l+1}^{(t)}}{\partial \boldsymbol{\theta}_l} = \widetilde{g}_l^{(t)}. \tag{14}$$

In our asynchronous system, however, to break the communication bottleneck, the target block l computes its surrogate local gradient $\widetilde{g}_l^{(t-\tau)}$ using stale forward activations and stale backward gradients delayed by τ steps. Thus, activations and gradients is mathematically equivalent to evaluating the loss using the τ-delayed weights of the preceding and succeeding blocks, while using the fresh weights for the local block l. Naturally, this allows us to rigorously define the surrogate local gradient as the true gradient evaluated on a *mixed-state parameter subspace* $\widetilde{\boldsymbol{\theta}}$ with each block $\widetilde{\boldsymbol{\theta}}_k$:

$$\widetilde{\boldsymbol{\theta}}_k = \begin{cases} \boldsymbol{\theta}_l^{(t)}, & \text{if } k = l \text{ (fresh local weights)} \\ \boldsymbol{\theta}_k^{(t-\tau)}, & \text{if } k \neq l \text{ (stale external signals)} \end{cases}$$

Thus, $\widetilde{g}_l^{(t-\tau)} \equiv \nabla_{\boldsymbol{\theta}_l} \mathcal{L}(\widetilde{\boldsymbol{\theta}})$. The asynchronous error can be explicitly defined as the distance between the gradient evaluated at the synchronous state $\boldsymbol{\theta}^{(t)}$ and the mixed state $\widetilde{\boldsymbol{\theta}}$. By the Mean Value Theorem and Assumption 4.1, we replace the maximum Hessian norm with the cross-block Lipschitz constant $H_{l,j}$:

$$\left\| \nabla_{\boldsymbol{\theta}_l} \mathcal{L}(\boldsymbol{\theta}^{(t)}) - \widetilde{g}_l^{(t-\tau)} \right\| \leq \sum_{j=1}^{L} H_{l,j} \left\| \boldsymbol{\theta}_j^{(t)} - \widetilde{\boldsymbol{\theta}}_j \right\|$$

Since the local block l inherently uses its most up-to-date weights during the surrogate local gradient computation, the parameter drift for block l is zero:

$$\left\|\boldsymbol{\theta}_l^{(t)} - \widetilde{\boldsymbol{\theta}}_l\right\| = \left\|\boldsymbol{\theta}_l^{(t)} - \boldsymbol{\theta}_l^{(t)}\right\| = 0$$

Consequently, the self-Hessian term $H_{l,l}$ is perfectly eliminated from the summation. The error is solely bounded by the cross-block Hessians ($j \neq l$):

$$\left\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}^{(t)}) - \widetilde{g}_l^{(t-\tau)}\right\| \leq \sum_{j\neq l} H_{l,j} \left\|\boldsymbol{\theta}_j^{(t)} - \boldsymbol{\theta}_j^{(t-\tau)}\right\| \tag{15}$$

Finally, using the triangle inequality and Assumption 4.1, we bound the accumulated parameter drift over the τ steps:

$$\left\|\boldsymbol{\theta}_j^{(t)} - \boldsymbol{\theta}_j^{(t-\tau)}\right\| \leq \sum_{k=1}^{\tau} \left\|\boldsymbol{\theta}_j^{(t-k+1)} - \boldsymbol{\theta}_j^{(t-k)}\right\| \leq \eta\tau G$$

Substituting this back into Eq. (15) and squaring both sides yields the final bound:

$$\left\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}^{(t)}) - \widetilde{g}_l^{(t-\tau)}\right\|^2 \leq \left(\eta\tau G \sum_{j\neq l} H_{l,j}\right)^2$$

This completes the proof. □

Proof of Theorem 4.4. From the Taylor Expansion result established in Equation 10, the loss descent for an individual block l is bounded by:

$$\mathcal{L}(\boldsymbol{\theta}_l^{(t+1)}) \leq \mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 - \left(\frac{\eta}{2} - \frac{L_l\eta^2}{2}\right)\|\tilde{g}_l^{(t-\tau)}\|^2 + \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \tilde{g}_l^{(t-\tau)}\|^2$$

Given the learning rate condition $\eta \leq \frac{1}{L_l}$, it follows that the coefficient of the third term is non-negative, meaning $\left(\frac{\eta}{2} - \frac{L_l\eta^2}{2}\right) \geq 0$. Therefore, we can drop the negative third term on the right-hand side to obtain a relaxed upper bound:

$$\mathcal{L}(\boldsymbol{\theta}_l^{(t+1)}) \leq \mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 + \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \tilde{g}_l^{(t-\tau)}\|^2$$

Substituting the asynchronous error bound derived in Lemma 4.3 into the above inequality, we have:

$$\mathcal{L}(\boldsymbol{\theta}_l^{(t+1)}) \leq \mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 + \frac{\eta}{2}\left(\eta\tau G \sum_{j\neq l} H_{l,j}\right)^2$$

Rearranging the terms to isolate the gradient norm yields:

$$\frac{\eta}{2}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 \leq \mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \mathcal{L}(\boldsymbol{\theta}_l^{(t+1)}) + \frac{\eta}{2}\left(\eta\tau G \sum_{j\neq l} H_{l,j}\right)^2$$

Multiplying both sides by $\frac{2}{\eta}$ gives:

$$\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 \leq \frac{2}{\eta}[\mathcal{L}(\boldsymbol{\theta}_l^{(t)}) - \mathcal{L}(\boldsymbol{\theta}_l^{(t+1)})] + \left(\eta\tau G \sum_{j\neq l} H_{l,j}\right)^2$$

Summing this inequality over $t = 0$ to $T-1$ and dividing by the total number of steps T leads to a telescoping sum on the loss evaluations:

$$\frac{1}{T}\sum_{t=0}^{T-1}\|\nabla_{\boldsymbol{\theta}_l}\mathcal{L}(\boldsymbol{\theta}_l^{(t)})\|^2 \leq \frac{2}{\eta T}[\mathcal{L}(\boldsymbol{\theta}_l^{(0)}) - \mathcal{L}(\boldsymbol{\theta}_l^{(T)})] + \eta^2\tau^2G^2\left(\sum_{j\neq l} H_{l,j}\right)^2$$

Since the loss is strictly bounded below by $\mathcal{L}^*$, we know that $\mathcal{L}(\boldsymbol{\theta}_l^{(0)}) - \mathcal{L}(\boldsymbol{\theta}_l^{(T)}) \leq \mathcal{L}(\boldsymbol{\theta}_l^{(0)}) - \mathcal{L}^*$. Substituting this absolute bound gives our final result:

$$\frac{1}{T}\sum_{t=0}^{T-1} \|\nabla_{\boldsymbol{\theta}_l} \mathcal{L}(\boldsymbol{\theta}^{(t)})\|^2 \leq \frac{2[\mathcal{L}(\boldsymbol{\theta}_l^{(0)}) - \mathcal{L}^*]}{\eta T} + \eta^2 \tau^2 G^2 \left(\sum_{j \neq l} H_{l,j}\right)^2$$

□

B Implementation Details

B.1 Model and Split Configuration

We fine-tune the Qwen2.5-1.5B-Instruct model Hui et al. (2024). The U-shaped split architecture is denoted by a triplet $[n_{\text{head}}, n_{\text{bulk}}, n_{\text{tail}}]$ giving the number of Transformer blocks on the client head, server bulk, and client tail, respectively. Throughout Section 5.2 we use the default split $[3, 22, 3]$. All methods employ a batch size of 16 with a cosine learning rate schedule.

B.2 Optimizer Details

Synchronous AdamW (baseline). We use the standard PyTorch implementation of AdamW Loshchilov & Hutter (2017) with β_1=0.9, β_2=0.999, weight decay 0.09, and ϵ=1×10^{-8}. The base learning rate is set to η=2×10^{-5} in the main experiments.

CommuteProp and CommuteProp (NS). CommuteProp wraps any base optimizer; the same AdamW hyperparameters serve the CommuteProp (AdamW) variant. The staleness delay τ is emulated via a queue-based simulator: a gradient update is triggered only after the corresponding forward activation has been queued for exactly τ micro-batches. For the utility comparison (Table 1) we set τ=16. The "NS" variant applies the Kronecker-Block Newton–Schulz preconditioner (Algorithm 3) before the AdamW step; we use a block size b=64 and refresh the preconditioner every 200 steps.

Nesterov Async. We re-implement the asynchronous pipeline parallel method of Ajanthan et al. (2025) with its default momentum factor 0.9 and Nesterov correction; the staleness delay is fixed at τ=16 for the fairness comparison.

Synchronous Muon (baseline). We use the standard PyTorch implementation of Muon Jordan et al. (2024) with β_1=0.9, β_2=0.999, weight decay 0.09, and ϵ=1×10^{-8}. The base learning rate is set to η=2×10^{-5} in the main experiments.

CommuteProp (Muon). The "Muon" variant replaces AdamW with the Muon optimizer Jordan et al. (2024) (momentum 0.95, Nesterov acceleration enabled) while keeping the CommuteProp two-phase pipeline unchanged.

B.3 Full-Training Comparison and Iso-time Comparison

In our experiments we report both full-training and iso-time comparisons. The full-training comparison verifies the utility of CommuteProp in end-to-end training, while the iso-time comparison, controlled by equal wall-clock budgets, directly captures the efficiency gain in the split learning scenario.

Our experiment use one single NVIDIA RTX A6000. For the iso-time comparison, we first profile the per-stage computation costs. We record the forward and backward times of the synchronous baseline (AdamW) and of CommuteProp (NS) over the first 50 training iterations and average the measurements. To isolate the speedup coming purely from computation-communication overlap, we exclude the time of optimizer updates, gradient zeroing, and learning rate scheduling, as these operations are nearly identical in both settings, non-overlappable, and would only add a constant offset to the end-to-end latency without affecting the relative gain from pipelining. Denote the $l = 1$ as the client node, $l = 2$ as the server node. The resulting mean per-stage costs are:

By using the Equation 6 and assuming one-way communication time is 0.05, we have $\kappa_a/\kappa_s = 1.92$; under an equal wall-clock budget the asynchronous method can process $1.92\times$ more data. Thus,

Table 3: Per-stage computation costs profiled over the first 50 iterations.

Method	Stage cost (s)	Notes
Sync baseline	$t_{1,s} + t_{2,s} = 0.089$	unified forward+backward
CommuteProp (NS)	Phase 1: $t_{1,1}, t_{2,1} = [0.0042,\ 0.0186]$ Phase 2: $t_{1,2}, t_{2,2} = [0.0364,\ 0.0695]$	separation of computation and update

under the iso-time constraint, the synchronous epoch budget is scaled by the inverse of the speedup. For example, if CommuteProp trains 5 epochs, under the time-constraint scenario, it trains for only $5/1.92 \approx 2.6$ epochs.

B.4 Client-server deployment experiment

To validate CommuteProp beyond the queue-based simulation, we conduct real distributed client–server experiments on two GPUs and use Linux `tc` to control the bandwidth (1 Gbps) and latency. Table 4 reports the per-iteration time of the synchronous baseline and CommuteProp under different latency settings. The measured speedups are consistently higher than the simulated estimates, confirming that the bubble-free synchronous simulation provides a conservative evaluation of CommuteProp's practical advantage.

Table 4: Real two-GPU client–server experiments under 1 Gbps bandwidth, with latency controlled by Linux `tc`.

Configured latency	**Measured latency**	**Sync/iter (s)**	**Async/iter (s)**	**Speedup**
10 ms	10.9 ms	0.155	0.068	2.28×
50 ms	51.0 ms	0.325	0.067	4.89×
100 ms	101.2 ms	0.534	0.066	8.07×
200 ms	201.4 ms	0.932	0.064	14.55×

To further benchmark CommuteProp against stronger system and scheduling baselines, we additionally include PipeDream-2BW, a weight-stashing pipeline method, and evaluate all methods under the same 15-minute wall-clock budget; the classification and generation results are reported in Table 5. CommuteProp achieves competitive or better performance than PipeDream-2BW under the same wall-clock budget, particularly when $\tau = 4$. We initially selected Nesterov Async as the primary asynchronous baseline because it avoids multi-version weight stashing and is therefore better suited to memory-constrained split-learning clients.

Table 5: Model utility under the 15-minute equal wall-clock budget. For each dataset–metric pair, the two columns correspond to $\tau = 4$ and $\tau = 16$.

	Classification (Acc)				**Generation**					
	Emotions		**AG-News**		**SAMSum**				**SciERC**	
	Acc		**Acc**		**ROUGE-L**		**METEOR**		**F1**	
Algorithm	$\tau=4$	$\tau=16$	$\tau=4$	$\tau=16$	$\tau=4$	$\tau=16$	$\tau=4$	$\tau=16$	$\tau=4$	$\tau=16$
Sync	0.7998	0.7378	0.8994	0.9120	0.3576	0.3585	0.3052	0.3042	0.3915	0.3805
CommuteProp	0.9076	0.9189	0.9220	0.9320	0.3704	0.3743	0.3203	0.3196	0.7747	0.7591
CommuteProp (NS)	0.8973	0.9097	0.9271	0.9300	0.3698	0.3707	0.3213	0.3183	0.7596	0.7814
CommuteProp (Muon)	0.9189	0.9251	0.8960	0.9120	0.3788	0.3999	0.3199	0.3402	0.7199	0.7868
PipeDream-2BW	0.9076	0.8819	0.9340	0.9060	0.3694	0.3697	0.3202	0.3159	0.7545	0.6743
Nesterov Async	0.6653	0.6181	0.8820	0.8800	0.3396	0.3323	0.2882	0.2782	0.0597	0.0623

B.5 Ablation Setup

The ablation study (Section 5.3, Figure 4) sweeps three factors while keeping all other settings at their default values:

1. **Staleness delay** τ: $\{4, 8, 16\}$ steps.
2. **Learning rate** η: $\{1 \times 10^{-5}, 2 \times 10^{-5}, 5 \times 10^{-5}\}$.
3. **Split architecture**: $[1, 26, 1]$, $[3, 22, 3]$

When not varied, the default values are τ=16, η=2×10^{-5}, and the $[3, 22, 3]$ split. Table 6 summarises the parameter combinations evaluated; any run not listed inherits the defaults described above.

Table 6: Grid of ablation settings. The default configuration (τ=16, η=2×10^{-5}, $[3, 22, 3]$) is highlighted and serves as the anchor for all single-factor sweeps.

Varied factor	Fixed factors	Values explored
Staleness τ	η=2×10^{-5}, split $[3, 22, 3]$	4, 8, **16**
Learning rate η	τ=16, split $[3, 22, 3]$	1×10^{-5}, $\mathbf{2\times10^{-5}}$, 5×10^{-5}
Architecture	τ=16, η=2×10^{-5}	$[1, 26, 1]$, $[\mathbf{3}, \mathbf{22}, \mathbf{3}]$

B.6 Datasets and Evaluation

We evaluate on four diverse NLP benchmarks: Emotion, AG-News, SAMSum, SciERC and Spider. Emotion Saravia et al. (2018) is a 6-class emotion classification dataset (sadness, joy, love, anger, fear, surprise) with 16,000 training samples and 2,000 test samples; texts are short social media posts with an average length of approximately 20 words. AG News Zhang et al. (2015) is a 4-class topic classification dataset (World, Sports, Business, Sci/Tech) with 120,000 training samples and 7,600 test samples; texts are news articles averaging 40–50 words. SAMSum Gliwa et al. (2019) is a dialogue summarization dataset with 14,732 training samples and 819 test samples; each dialogue contains multiple turns of casual conversation, and the reference summaries average approximately 20 words. SciERC Luan et al. (2018) is a scientific relation extraction dataset with 3,219 training samples and 974 test samples; texts are scientific paper abstracts, and the task is to classify the relation between annotated entity pairs into one of 7 relation types (labels 0–6). Spider Yu et al. (2018) is a cross-domain text-to-SQL dataset with 7,000 training samples and 1,034 test samples; each example provides a natural language question paired with the corresponding SQL query spanning 200 databases. For all datasets, we report final results on the official test split. Emotion, AG-News and Spider are trained for 3 epochs, while SAMSum and SciERC are trained for 5 epochs.

For preprocessing, we apply the Qwen2.5-1.5B-Instruct chat template to format each example into a consistent instruction-following format, and truncate all sequences to a maximum length of 2,048 tokens. For Spider, the instruction asks the model to translate the question into an SQL query. Padding is performed dynamically per batch, which also masks the user-instruction portion of each sequence in the loss computation using `labels=-100`, so that the model is trained only on the assistant response tokens.

Reported metrics are Accuracy (for classification), ROUGE-L and METEOR (for summarization), macro-F1 (for relation extraction), and exact-set-match (EM) accuracy (for text-to-SQL), consistent with prior work.

For Emotion (6-class emotion classification) and AG-News (4-class topic classification), we compute accuracy as the fraction of samples where the model's predicted label exactly matches the ground-truth label. For SAMSum (dialogue summarization), we report ROUGE-L F-measure as the primary metric and METEOR as a secondary metric; both are computed using the HuggingFace `evaluate` library with default settings (ROUGE-L with stemming enabled via `use_stemmer=True`). For SciERC (scientific relation extraction with 7 relation types), we report macro-averaged F1 score across all 7 classes. For Spider, we report exact-set-match accuracy, which compares the execution results of the predicted SQL query against the gold query on the target database—a prediction is considered correct only if the two result sets are identical.

During inference, we use greedy decoding (`do_sample=False`) with dataset-specific maximum generation lengths: 10 tokens for Emotion and AG-News (single-label output), 20 tokens for SciERC (single-digit output), 128 tokens for Spider (SQL query generation), and 128 tokens for SAMSum (summary generation).

B.7 Integration with DualGuard

B.7.1 Introductions

In Section 5.3. We integrate CommuteProp into the DualGuard framework Liu et al. (2025). DualGuard assumes that the adversaries can access to (1) the pre-trained model weights (publicly available), (2) the transmitted smashed data and gradients during training, and (3) an auxiliary dataset for training attack models. The attacker aims to reconstruct the private training data. Specifically, they introduce four attack mechanisms:

SIP(Smashed-data Inversion based on Pre-trained model): The attacker trains an inversion model (a GRU-based decoder followed by an MLP projection) on an auxiliary dataset using the pre-trained head model. The inversion model learns to map smashed data (head model activations) back to input token sequences.

TAG (Text Revealer from Gradients): TAG reconstructs output labels by matching gradients. The attacker intercepts the gradients flowing back from the tail model to the server. Using the pre-trained tail model, the attacker initializes dummy label logits and iteratively optimizes them to minimize the distance between the dummy gradients (produced by the pre-trained tail on the dummy labels) and the true gradients. The recovered labels in LLM tasks substantially overlap with the input, enabling reconstruction of the private sequence.

LAMP (Label-extracting Attack with Model Priors): LAMP extends TAG by incorporating stronger language model priors. Instead of optimizing over continuous logit space, LAMP alternates between (1) discrete token selection based on the current dummy logits and (2) gradient matching optimization.

CPTA (Connecting Pre-trained Tail Attack): CPTA directly connects the pre-trained tail model to the server's output hidden states, bypassing the trained tail model entirely. The attacker simply feeds the intercepted server outputs into the publicly available pre-trained tail model and evaluates its generation against the private labels.

DualGuard Defense. DualGuard defends against these attacks through two stages:

Stage 1: Local Warm-Up Parameter Space Transformation. Before formal fine-tuning, a three-layer projection MLP is inserted between the head and tail models. The head, tail, and ProjectionMLP are jointly optimized on the client side via multi-task learning with three objectives: (1) maximizing the SIP inversion loss to disrupt smashed-data reconstruction, (2) maximizing the pre-trained tail's cross-entropy loss to deactivate gradient matching, and (3) minimizing the standard LM loss to preserve task performance. LoRA adapters are used for parameter-efficient training on the resource-constrained client. The trunk model is not involved in this stage.

Stage 2: Global Fine-Tuned Parameter Space Retention. During formal split-based fine-tuning, the head model is frozen to maintain the secure forward activations established in Stage 1. LoRA adapters on the server (trunk) and tail models are fine-tuned on the target dataset, while base model weights remain frozen. An anti-pre-trained-tail loss term ($\lambda_3/\mathcal{L}_{\text{global-pt}}$) is added to prevent the tail from reverting to its vulnerable pre-trained parameter space. In our experiments, we replace the original synchronous training algorithm in Stage 2 with our CommuteProp algorithm.

B.7.2 HyperParameters

Our hyper-parameters are nearly the same as DualGuard. For split architecture, the original paper claims the $[2, 12, 2]$ for Qwen2-1.5B, which does not match the total block numbers(28). Thus, we replace it with [3,22,3]. Full hyper-parameters are detailed in Table 7.

Table 7: Hyperparameter configuration for DualGuard + CommuteProp experiments.

Setting	Hyperparameters
Model & Data	base = Qwen2-1.5B, split = [3,22,3], max_seq_len = 512, dataset = GSM8kCobbe et al. (2021)
LoRA	$r = 2$, $\alpha = 32$, dropout = 0, targets = {q_proj, v_proj}
Warm-Up	epoch = 2, bs = 4, lr = 2×10^{-5}, AdamW, wd = 0.01, $\lambda_1 = 40$, $\lambda_2 = 80$
Fine-Tuning	epoch = 4, bs = 4, lr = 2×10^{-5}, AdamW, wd = 0.01, $\lambda_3 = 10$, $\tau = 4$
Attack Eval.	samples = {SIP: 131, TAG/LAMP: 40}, iters = {TAG: 500, LAMP: 200 ($\times$30 inner)}, PT-tail wt = 70.0

B.7.3 Experiment Result

In the *LoRA* setting, LoRA adapters (r=2) are injected into all model segments; only adapter weights on the server and tail are updated while base parameters remain frozen. In the *full-parameter* setting, the server is trained without LoRA—all base weights are updated—while head and tail remain LoRA-based from Stage 1.

Table 8: Performance comparison under different attack methods.

	Performance (↑)		Attack (Rouge-L-F %, ↓)			
Method	Rouge-L	METEOR	SIP	TAG	LAMP	CPTA
w/o defense (naive USL)	86.23	88.48	86.87	65.70	63.13	17.75
Full-param DualGuard (sync)	85.76	88.13	5.64	38.67	38.26	0.53
Full-param CommuteProp (τ=4)	84.93	87.51	5.64	9.45	11.54	0.00
Full-param CommuteProp(NS)	84.89	87.25	5.64	23.15	23.21	5.43
LoRA DualGuard (sync)	85.12	86.92	5.64	5.06	5.81	3.54
LoRA CommuteProp (τ=4)	85.18	87.13	5.64	4.34	4.57	3.58
LoRA CommuteProp(NS)	85.29	87.17	5.64	6.66	6.20	6.76

The results indicate that CommuteProp achieves stronger privacy protection than synchronous DualGuard with negligible utility degradation. We attribute this improvement to the implicit regularization induced by stale activations: the temporal offset between the head output and the server's computation introduces a controlled distributional shift, further decorrelating the server's output from the pre-trained model's expectations and thereby weakening gradient-matching attacks.

C Additional Results

C.1 Addition Experiment Result on Different Base-models

Setup We additionally run experiments with a larger base model, Llama3.2-3B-InstructGrattafiori et al. (2024), keeping the same U-shaped split [3, 22, 3] and all hyper-parameters from Appendix B (batch size 16, cosine learning rate schedule, etc.). To stabilise asynchronous training on this larger model, we introduce gradient clipping with a maximum global norm of 1.0. Using the same profiled one-way communication time and technique in Section B.3, we obtain a speed-up factor of $1.98\times$.

Table 9: Per-stage computation costs profiled over the first 50 iterations For Llama3.2-3B model

Method	Stage cost (s)	Notes
Sync baseline	$t_{1,s} + t_{2,s} = 0.107$	unified forward+backward
CommuteProp (NS)	Phase 1: $t_{1,1} + t_{2,1} = [0.0041, 0.0207]$ Phase 2: $t_{1,2}, t_{2,2} = [0.0406,\ 0.0701]$	separation of computation and update

Results Due to limited computational resources and time constraints, we were unable to exhaustively evaluate the full grid of baselines and tasks. For the experiments we conducted, we set the number of training epochs to 1 for AGNews. The results, reported in Table 10, — consistently demonstrates that CommuteProp (with NS preconditioner) preserves utility close to the synchronous AdamW baseline while effectively hiding communication latency.

Table 10: Utility comparison under full-training and Equal time budgets

Algorithm	Emotion (Acc)	AGNews (Acc)	SciERC (F1 Score)
Zero-shot	40.8	68.00	4.55
AdamW	93.75	96.50	82.94
Nadam	65.25	88.12	12.84
AdamW (Time constraint)	**93.65**	95.00	76.84
CommuteProp (NS)	93.41	**95.21**	**83.40**

C.2 Empirical Validation of Cross-Block Hessian Structure

To empirically examine Assumption 4.1, we directly measure the Hessian statistics of the evaluated split architectures, including Qwen2.5-1.5B. Table 11 reports the element-wise statistics of the diagonal (within-block) and cross-block Hessian entries. The results show that cross-block interactions are substantially weaker than the dominant within-block terms, supporting the hierarchical smoothness assumption used in our analysis. This observation is also consistent with prior studies on the approximately block-diagonal Hessian structure of large neural networks Zhang et al. (2024); Dong et al. (2025).

Following the empirical protocols in Zhang et al. (2024); Dong et al. (2025), we use element-wise Hessian statistics to support the structural assumption rather than directly estimating the block operator norm, whose computation is prohibitive at the billion-parameter scale. We emphasize that our convergence and performance results are obtained from full-training runs and are independent of this estimate.

Table 11: Measured element-wise Hessian statistics on the evaluated Qwen2.5-1.5B split architectures.

Statistic	Diagonal	Cross-block	Cross/Diagonal
Max	1.49×10^{-2}	7.72×10^{-3}	0.39
P99	6.09×10^{-3}	3.11×10^{-4}	0.05
Median	4.05×10^{-5}	2.81×10^{-6}	0.07