FRESCO: A Novel Consistency Control for Asynchronous Pipeline Parallel Training

Dongyeop Lee
POSTECH
dongyeop.lee2@postech.ac.kr

Namhoon Lee
POSTECH
namhoon.lee@postech.ac.kr

Abstract

Asynchronous pipeline parallelism can accelerate distributed model training, but suffers from instability caused by inconsistent gradient computations. We identify a fundamental source of this inconsistency: error signals are coupled to forward-time features, not just parameter states. This means that parameters may evolve as long as they preserve *forward-equivalence*. Leveraging this insight, we introduce FRESCO, which frames asynchronous updates as a constrained optimization problem: it finds the minimal update modification that acts trivially on the activation subspace. We demonstrate that FRESCO consistently outperforms representative asynchronous baselines in LLM pre-training, with the gains most pronounced under deep pipelines and high inconsistency levels: at pipeline depth 32, it reaches the same loss as the strongest baseline with 43% fewer iterations and 30% less wall-clock time, while requiring 8–$921\times$ less memory overhead. Moving beyond the current limitations, FRESCO establishes a new foundation for high-utilization pipeline parallel training, providing a scalable path for large-model development on diverse, resource-constrained systems.

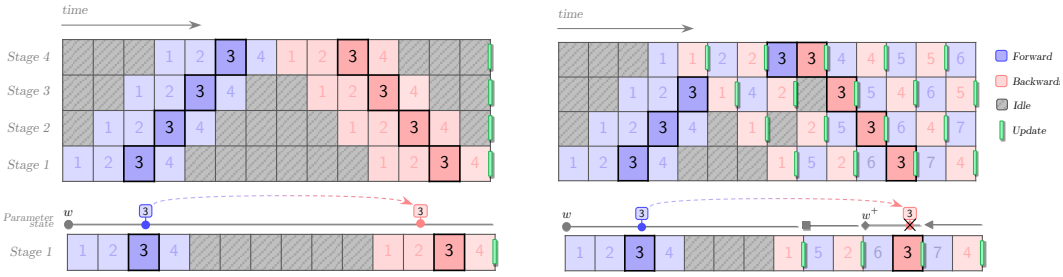
1 Introduction

Pipeline parallelism has played a central role in the scaling of modern machine learning by partitioning enormous neural networks across multiple processing units along their sequential data-processing structure, enabling large-scale distributed training across hundreds or even thousands of devices [10, 29, 43]. Pipeline scheduling has been integral to fully exploiting the computational capabilities of these systems, continuously streaming micro-batches through successive stages to keep different processing units concurrently occupied rather than idle [35, 27, 16, 22, 21, 32].

Asynchronous pipelining offers a straightforward way to achieve sustained, bubble- and halt-free pipeline execution by allowing parameter updates to proceed concurrently with the forward and backward computations of in-flight micro-batches (see Figure 1 for illustrations). However, lifting this parameter-equivalence condition can introduce two types of inconsistencies into these concurrent gradient computations: *chain inconsistency*, where partial gradients are chained across different parameter versions, and *gradient staleness*, where even a chain-consistent gradient no longer corresponds to the current parameter state, both of which can degrade training stability and convergence. To mitigate these adverse effects, various correction mechanisms have been developed to compute gradients with respect to either past parameter versions, obtained through stashing or reconstruction [32, 33, 47], or predicted future parameter states [5, 18, 19].

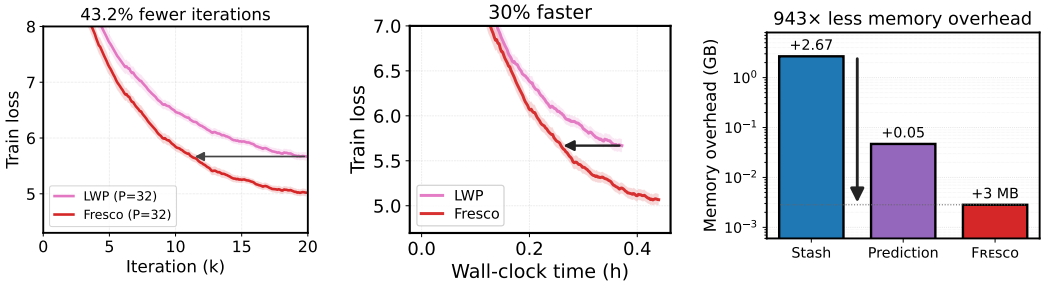
Yet, prior methods have largely overlooked the possibility of controlling parameter updates themselves to preserve concurrent gradient computations consistent, even though preventing interference with these computations was the very motivation for imposing the restriction in the first place. Consequently, asynchronous pipeline optimization has yet to establish what principle should replace the parameter-equivalence condition that safeguards gradient computations in synchronous execution. In-

40th Conference on Neural Information Processing Systems (NeurIPS 2026).



(a) Synchronous pipeline parallelism (b) Asynchronous pipeline parallelism

Figure 1: Gantt charts illustrating stage-processor states over time for micro-batch pipelining under synchronous and asynchronous parameter updates. Asynchronous updating allows the stage-1 threads for micro-batches 1 and 2 to overwrite the parameters while micro-batch 3 is still waiting on its downstream error signal ● after reading the previous parameter state ●, thus the updated parameters ◆ no longer form a proper derivative chain, degrading its utility for improving stage 1.



(a) Robust under deep pipeline (b) Fast training (c) Memory efficiency

Figure 2: Summary of results. (a) Under a deep asynchronous pipeline, FRESCO improves optimization over best baseline, reaching the same training loss with fewer iterations, (b) yielding faster training speeds under wall-clock time. (c) FRESCO avoids the large memory overhead of maintaining multiple full parameter versions, using substantially less memory than stashing-based approaches.

stead, existing methods have primarily focused on compensating post hoc for errors within individual micro-batches, rather than controlling the inter-micro-batch interference that gives rise to them. Simply lifting the restriction to permit unconstrained parameter updates provides no principled criterion for determining which changes can preserve gradient consistency without introducing staleness.

In this work, we address this gap by showing that parameter changes degrade gradient computation not simply by altering the parameters, but by breaking the compatibility between a returned error signal and the forward behavior of the updated stage at the inter-stage boundary. Concretely, when the updated parameters would produce a different stage output from the one on which the error signal was evaluated, the error signal and the local Jacobian no longer form a valid gradient for the updated parameters. This reveals that the relevant consistency requirement is preserving the forward behavior seen by in-flight micro-batches even if the full parameter tensor changes, *i.e.*, *forward-equivalence* over parameter-equivalence.

Guided by this insight, we propose FRESCO, an asynchronous pipeline-parallel training method that controls parameter updates according to their effect on the forward activations of in-flight micro-batches. Concretely, we formulate this as a constrained optimization problem that finds the closest modification of each optimizer update subject to acting trivially on the activation subspace occupied by in-flight micro-batches. To implement this efficiently, FRESCO maintains a compact spectral basis for this activation subspace and projects each layer-wise update onto its orthogonal complement, thereby removing exactly the components that would violate forward equivalence. This allows parameters to evolve asynchronously while preserving the forward behavior required for the returned error signals to remain valid. Finally, we establish convergence of FRESCO, showing that it control the degree of perturbation introduced during forward and backwards pass to guarantee convergence.

Empirically, FRESCO improves asynchronous pipeline-parallel training across model scales and pipeline depths, achieving lower loss than representative asynchronous baselines in an established pipeline-parallel pre-training setting with a 95M-parameter language model, especially at moderate and deep pipeline depths (Figure 2a), allowing FRESCO to achieve the same loss with 30% less wall-clock time (Figure 2b). Moreover, because FRESCO caches only compact dominant feature spectra rather than full parameter copies, its memory overhead remains marginal compared with existing baselines, especially for larger models (Figure 2c).

These results indicate that FRESCO provides an effective foundation for reliable, high-utilization pipeline-parallel training, opening a new avenue beyond strict parameter consistency toward scalable training on conventional and resource-constrained collaborative systems.

2 Problem analysis

We begin by presenting a detailed formulation of the parameter inconsistency problem in asynchronous pipeline-parallel training and the forward-equivalence condition.

Preliminary A pipeline-parallel training setup partitions a model $F = f_P \circ \cdots \circ f_1$ across multiple processing units, each responsible for a stage f_p. Each stage processes a stream of micro-batches, typically chosen as the largest units that fit in device memory. For simplicity, we describe the computation from the perspective of a generic stage f with parameters $\boldsymbol{w}$ and a single micro-batch, represented by its input and output features $\boldsymbol{s}$ and $\boldsymbol{s}_+$, omitting stage and micro-batch indices. We also denote the function that takes the input feature $\boldsymbol{s}$ and evaluate the loss value (*i.e.*, the composition of downstream stages and the loss function) as simply $L(\boldsymbol{s}_+) \in \mathbb{R}$. At a high level, each stage repeatedly performs three operations for each micro-batch: it computes the forward signal $\boldsymbol{s}_+ = f(\boldsymbol{w}, \boldsymbol{s})$ and sends it downstream; it receives a downstream error signal $\boldsymbol{e}_+$ and computes the backward signal $\boldsymbol{e}$ together with the local parameter gradient $\boldsymbol{g}$; and, in between, it waits for downstream stages to complete the dependent forward or backward computations.

A close look at asynchronous pipelining Writing the asynchronous pipeline computation for a generic stage and a single micro-batch reveals its characteristic structure:

$$\boldsymbol{s}_+ = f(\boldsymbol{w}, \boldsymbol{s}) \quad \triangleright \textit{ Forward signal} \tag{1}$$
$$(\boldsymbol{w} \to \boldsymbol{w}^+, \text{ wait for } \partial_{\boldsymbol{s}_+} L) \quad \triangleright \textit{ Asynchronous updates} \tag{2}$$
$$\boldsymbol{e}_+ = \partial_{\boldsymbol{s}_+} L(\boldsymbol{s}_+) \quad \triangleright \textit{ Returned error signal} \tag{3}$$
$$\boldsymbol{e} = \partial_{\boldsymbol{s}} f(\boldsymbol{w}^+, \boldsymbol{s})^*[\boldsymbol{e}_+] \quad \triangleright \textit{ Backprop error signal} \tag{4}$$
$$\boldsymbol{g} = \partial_{\boldsymbol{w}} f(\boldsymbol{w}^+, \boldsymbol{s})^*[\boldsymbol{e}_+] \quad \triangleright \textit{ Compute gradient} \tag{5}$$
$$\Delta = -U(\boldsymbol{g}) \quad \triangleright \textit{ Compute update}, \tag{6}$$

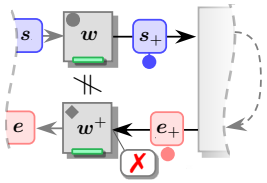


Figure 3: Parameter-inconsistent gradient computation.

where $\partial_{\boldsymbol{s}} f(\boldsymbol{w}^+, \boldsymbol{s})^*[\boldsymbol{e}_+]$ and $\partial_{\boldsymbol{w}} f(\boldsymbol{w}^+, \boldsymbol{s})^*[\boldsymbol{e}_+]$ denote the reverse-mode chain-rule operations.

One property exposed by this formulation is how inconsistency becomes more severe for earlier pipeline stages. The wait in (2) accumulates upstream: before an upstream stage receives its returned error signal, the neighboring downstream stage must itself complete its waiting interval, together with the additional forward and backward computations in (1–5). As shown in Figure 1, this interval is long enough for the upstream stage to process other micro-batches and apply intervening updates, so the parameter state may drift from $\boldsymbol{w}$ to $\boldsymbol{w}^+$ before the original micro-batch returns for backpropagation in (3), illustrated in Figure 3.

Forward-equivalence condition Attending to the gradient computation in (5), we identify the inconsistency between the returned error signal $\boldsymbol{e}_+$ and the parameter state used in the local Jacobian $\partial_{\boldsymbol{w}} f(\boldsymbol{w}^+, \boldsymbol{s})$, which generally prevents them from forming a proper derivative chain. This raises a natural question: *can the same error signal $\boldsymbol{e}_+$ remain valid across different parameter states $\boldsymbol{w}^+$ that induce different local Jacobians?* The key observation is that $\boldsymbol{e}_+$ depends on the downstream feature $\boldsymbol{s}_+$, not directly on the parameter tensor $\boldsymbol{w}$ itself. Therefore, if the updated parameter $\boldsymbol{w}^+$

produces the same forward feature for the in-flight micro-batch, then the returned error signal can still be chained with the local Jacobian at $\boldsymbol{w}^+$:

$$\forall \boldsymbol{w}^+ : f(\boldsymbol{w}^+, \boldsymbol{s}) = f(\boldsymbol{w}, \boldsymbol{s}) = \boldsymbol{s}_+ \quad \Longrightarrow \quad \boldsymbol{g} = \partial_{\boldsymbol{w}} f(\boldsymbol{w}^+, \boldsymbol{s})^*[\boldsymbol{e}_+] \text{ is a proper gradient at } \boldsymbol{w}^+$$

This forward-equivalence condition is the key idea behind our work: the full parameter tensor need not remain fixed, as long as the stage preserves the forward behavior seen by the in-flight micro-batch until its error signal returns. Under this condition, the returned error signal and the changed local Jacobian still form a valid gradient at the updated parameter state. The remaining question is therefore how to constrain asynchronous parameter updates so that they preserve these boundary signals while still allowing the parameters to evolve.

3 Method

Building on this insight, we design Forward-Equivalent Subspace-Constrained Optimization, or FRESCO, as the solution to the constrained optimization problem described below. The resulting update rule exploits a natural non-interference structure of linear operators: updates that annihilate the in-flight activation subspace do not change the corresponding forwarded features.

3.1 Derivation

Consider a linear layer with bias absorbed into the weight matrix: $Y = W\bar{X}$, where $W \in \mathbb{R}^{d_{\text{out}} \times (d_{\text{in}}+1)}$. For each such layer, we solve the constrained optimization problem

$$\min_{\Delta'} \frac{1}{2}\|\Delta' - \Delta\|_F^2 \quad \text{s.t.} \quad (W + \Delta')\bar{X} = W\bar{X}, \tag{7}$$

where $\bar{X} \in \mathbb{R}^{(d_{\text{in}}+1)\times \tau n}$ denotes the concatenated input features of all τ in-flight micro-batches, each of size n, whose forward computations are to be protected from asynchronous parameter updates $\Delta \in \mathbb{R}^{d_{\text{out}} \times (d_{\text{in}}+1)}$. Thus, the goal is to construct the feasible update Δ' closest in Frobenius norm $\|\cdot\|_F$ to the original update Δ such that the updated weight $W + \Delta'$ preserves the same forward pass as W on the protected micro-batch. If every asynchronous update that intersects the lifetime of this micro-batch is subjected to the same constraint, then the final updated weight W^+ remains forward-equivalent to the weight state under which the micro-batch entered protection.

Rewriting the preservation condition as

$$\Delta'\bar{X} = 0, \tag{8}$$

shows that the constraint does not explicitly depend on the weight W; it only depends on the action of the update on the protected in-flight activations.

We solve the constrained problem through the KKT conditions. Introducing a matrix-valued Lagrange multiplier $\Lambda \in \mathbb{R}^{d_{\text{out}} \times \tau n}$ with the same shape as $\Delta'\bar{X}$, the Lagrangian is

$$\mathcal{L}(\Delta', \Lambda) = \frac{1}{2}\|\Delta' - \Delta\|_F^2 + \langle \Lambda, \Delta'\bar{X}\rangle_F,$$

where $\langle A, B\rangle_F = \operatorname{tr}(A^\top B)$ denotes the Frobenius inner product. Because the objective is convex quadratic and the constraint is affine, the KKT conditions are necessary and sufficient:

$$\Delta'\bar{X} = 0, \qquad \nabla_{\Delta'}\mathcal{L}(\Delta', \Lambda) = 0. \tag{9}$$

The stationarity condition in Equation (9) gives

$$0 = \Delta' - \Delta + \Lambda\bar{X}^\top, \qquad \Delta' = \Delta - \Lambda\bar{X}^\top. \tag{10}$$

Substituting Equation (10) into the feasibility condition gives $0 = (\Delta - \Lambda\bar{X}^\top)\bar{X} = \Delta\bar{X} - \Lambda\bar{X}^\top\bar{X}$, and therefore $\Lambda\bar{X}^\top\bar{X} = \Delta\bar{X}$. Using the Moore–Penrose pseudoinverse[1], we write $\Lambda = \Delta\bar{X}(\bar{X}^\top\bar{X})^\dagger$. Finally, substituting this into Equation (10) yields

$$\Delta'_\star = \Delta\left(I - \bar{X}(\bar{X}^\top\bar{X})^\dagger\bar{X}^\top\right). \tag{11}$$

[1] Although Λ may not be unique when $\bar{X}^\top\bar{X}$ is singular, all valid choices yield the same product $\Lambda\bar{X}^\top$, and hence the same primal solution $\Delta'_\star$.

Here, the factor $\bar{X}(\bar{X}^\top \bar{X})^\dagger \bar{X}^\top$ in Equation (11) is the orthogonal projector onto $\mathrm{col}(\bar{X})$. Using the compact SVD $\bar{X} = U\Sigma V^\top$, we can write $P_{\bar{X}} := \bar{X}(\bar{X}^\top \bar{X})^\dagger \bar{X}^\top = UU^\top$. Thus,

$$\Delta'_\star = \Delta(I - P_{\bar{X}}) = \Delta(I - UU^\top). \tag{12}$$

3.2 Relaxation

For extremely deep pipelines, concurrency can reach a regime where each parameter update affects an increasing number of in-flight micro-batches, posing distinct challenges to both existing methods and FRESCO. While existing methods suffer from increasingly stale gradients or unreliable predictions of future parameter states, FRESCO may find that protecting all affected micro-batches leaves no permissible direction for parameter movement, potentially annihilating the update altogether, *i.e.*, $\Delta'_\star = 0$.

Existing correction-based methods are not inherently equipped to handle this regime and often resort to auxiliary mechanisms such as step-size damping or momentum to maintain stability [47, 1]. Unlike these methods, FRESCO offers an explicit trade-off between gradient fidelity and effective parameter movement by relaxing the protection condition to preserve the most influential directions while allowing greater freedom for asynchronous parameter updates.

We do this by protecting only the spectrally dominant components of $\mathrm{col}(\bar{X})$. Specifically, we construct a rank-k protection matrix $\bar{X}_k$ as the best low-rank approximation of $\bar{X}$ in the Frobenius norm: $\bar{X}_k = \arg\min_{\mathrm{rank}(A)\le k} \|\bar{X} - A\|_F$. Substituting this approximation into FRESCO yields the projection matrix $P_k = U_k U_k^\top$, where U_k contains the top-k left singular vectors of $\bar{X}$. The resulting rank-limited update is

$$\Delta'_k = \Delta(I - U_k U_k^\top). \tag{13}$$

As the rank budget k increases, the relaxation approaches exact preservation; when $k = \mathrm{rank}(\bar{X})$, Equation (13) recovers the exact projection in Equation (12).

Since weight matrices vary in dimensions d_{out} and d_{in}, their protection capacities $\min(d_{\text{out}}, d_{\text{in}} + 1)$ also differ. Using a fixed rank across layers may therefore lead to disproportionate protection or unnecessarily restrict parameter updates. Instead, we allocate a fraction $\alpha \in [0, 1]$ of each layer's capacity, capped by the maximum number τn of in-flight feature directions:

$$k = \min\left\{\tau n,\ \lceil \alpha \min(d_{\text{out}}, d_{\text{in}} + 1)\rceil\right\}. \tag{14}$$

This provides a consistent relative protection budget across layers without allocating more directions than the in-flight features can span. In practice, U_k is assembled from per-micro-batch bases rather than computed directly from $\bar{X}$, as described in Section 3.3.

3.3 Algorithm

Here, we describe the algorithmic operation at a high level; the full procedure is given in Algorithm 1. For each linear layer, FRESCO maintains a FIFO cache of low-rank activation bases for the in-flight micro-batches. Since the protected rank k is shared among the τ in-flight micro-batches, each micro-batch contributes $k_1 = \lceil k/\tau \rceil$ directions, so that the cache holds $\tau k_1 \ge k$ candidate directions. When a new micro-batch enters the layer, FRESCO calls EXTRACTBASIS$(\bar{X}, k_1)$ to compute a rank-k_1 basis of its bias-augmented input activations and enqueues the result. We use an inexpensive approximation: a single step of block power iteration, orthogonalized by a single Newton–Schulz step (Appendix C.3). When the corresponding backward error signal returns, that basis is dequeued, since the micro-batch no longer needs protection.

Before applying an optimizer update, FRESCO calls BUILDPROTECTEDBASIS$(\mathcal{Q}, k)$, which concatenates the queued bases and keeps their dominant k directions by block power iteration. This produces a single orthonormal basis $Q \in \mathbb{R}^{(d_{\text{in}}+1)\times k}$, the rank-$k$ protected subspace shared by the in-flight micro-batches. The base optimizer first proposes the usual update Δ; FRESCO then applies $\Delta' = \Delta(I - QQ^\top)$ instead, removing only the components that would change the protected forward features of the still in-flight micro-batches. Because each micro-batch stores only $\lceil k/\tau \rceil$ directions, the cache holds fewer than $k + \tau$ basis vectors per layer regardless of the pipeline depth, and it stores only compact activation bases rather than delayed weight copies or predicted parameter buffers. Thus, the main overhead is computation for subspace extraction, basis construction, and projection, rather than model-scale memory.

Algorithm 1 FRESCO step

Require: Weight $W \in \mathbb{R}^{d_{\text{out}} \times (d_{\text{in}}+1)}$ (bias absorbed), input activation X, gradient G, protection budget α, number τ of micro-batches in flight at this stage, mode $\in \{\text{FORWARD}, \text{BACKWARD}\}$
State: FIFO queue $\mathcal{Q}$ storing protected activation bases

if $\mathcal{Q}$ is uninitialized **then**
  Initialize $\mathcal{Q} \leftarrow \emptyset$
end if
if *mode* $=$ FORWARD **then**
  $\bar{X} \leftarrow [X; \mathbf{1}^\top] \in \mathbb{R}^{(d_{\text{in}}+1) \times n}$ ▷ $n =$ batch $\times$ sequence
  $k \leftarrow \min(\tau n, \lceil \alpha \min(d_{\text{out}}, d_{\text{in}}+1) \rceil)$ ▷ protected rank
  $k_1 \leftarrow \lceil k/\tau \rceil$ ▷ rank per micro-batch
  $V \leftarrow \text{EXTRACTBASIS}(\bar{X}, k_1)$ ▷ $V \in \mathbb{R}^{(d_{\text{in}}+1) \times k_1}$
  $\mathcal{Q}.\text{ENQUEUE}(V)$
  return
end if
if *mode* $=$ BACKWARD **then**
  $\Delta \leftarrow \text{OPTIMIZERUPDATE}(W, G)$ ▷ base optimizer proposes update
  $\mathcal{Q}.\text{DEQUEUE}()$ ▷ remove the completed micro-batch basis
  $Q \leftarrow \text{BUILDPROTECTEDBASIS}(\mathcal{Q}, k)$ ▷ top-k subspace of the remaining bases
  $\Delta' \leftarrow \Delta(I - QQ^\top)$ ▷ project away feature-changing directions
  $W \leftarrow W + \Delta'$ ▷ apply forward-equivalent update
end if

4 Theory

We provide a preliminary convergence analysis of FRESCO under asynchronous pipeline execution. Building on the perturbed forward–backward SGD framework of Kong et al. [25], we examine how the forward perturbations induced by concurrent parameter updates contribute to gradient errors and convergence.

For a linear stage, the residual forward perturbation of a projected update is

$$R_{i,t} := \|\Delta W'_{i,t} \bar{S}_{i,t}\|_F, \tag{15}$$

where $\bar{S}_{i,t}$ contains the protected in-flight features. For rank-k projection, this residual is bounded by

$$R_{i,t} \le \sigma_{i,t,k+1} \|\Delta W'_{i,t}\|_F, \tag{16}$$

which vanishes when the complete activation subspace is protected. Under bounded-delay execution, these residuals can be related to asynchronous gradient errors through the perturbation bounds of Kong et al. [25].

To examine the convergence implications, we consider a soft extension of FRESCO,

$$\Delta W'_{i,t} = \Delta W_{i,t} A_{i,t}, \qquad A_{i,t} := (I + \lambda_i U_{i,t,k} \Sigma^2_{i,t,k} U^\top_{i,t,k})^{-1}, \tag{17}$$

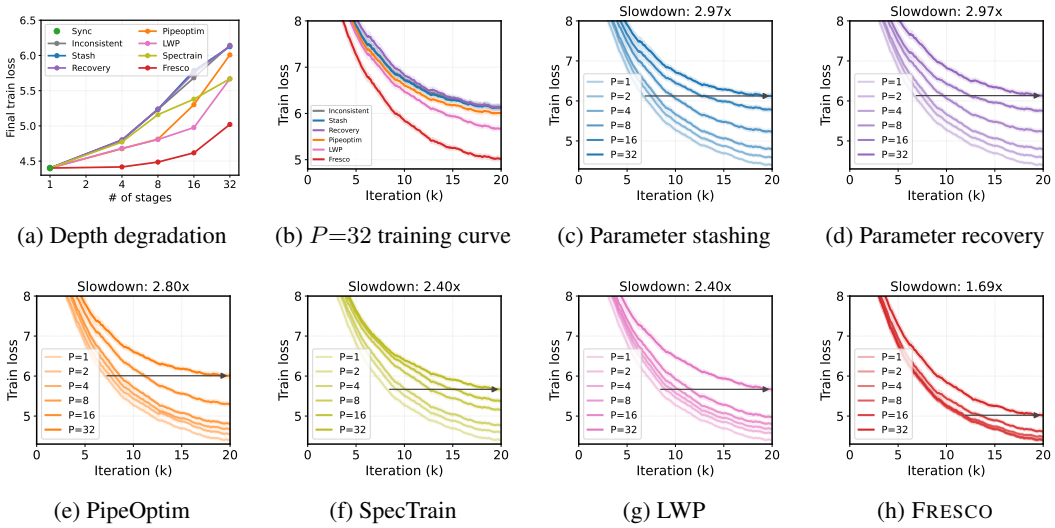
which approaches hard projection as $\lambda_i \to \infty$.

Theorem 4.1 (Convergence under soft projection)**.** *Under the assumptions in Appendix B, consider the update $\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \mathcal{A}_t \tilde{\boldsymbol{g}}_t$, where $\mathcal{A}_t$ collects the soft projection operators and satisfies $aI \preceq \mathcal{A}_t \preceq I$ for some $a > 0$. For $\eta = c/\sqrt{T}$ with $0 < c \le 1/(4L)$,*

$$\frac{1}{T} \sum_{t=0}^{T-1} \mathbb{E} \|\nabla \mathcal{L}(\boldsymbol{w}_t)\|^2 \le \frac{2\Delta_{\mathcal{L}} + 4Lc^2\sigma^2}{ac\sqrt{T}} + \frac{3c^2(K_{\text{f}} s^2 + K_{\text{b}})}{aT}, \tag{18}$$

where $\Delta_{\mathcal{L}} := \mathcal{L}(\boldsymbol{w}_0) - \mathcal{L}_\star$, s bounds the residual forward sensitivity, and $K_{\text{f}}, K_{\text{b}}$ depend on the computational graph and pipeline delays.

This suggests that controlling forward perturbations is compatible with the standard $O(T^{-1/2})$ stochastic convergence rate under the stated conditions. The guarantee is limited to the simplified soft-projection model and does not establish convergence acceleration or a full-gradient guarantee for hard FRESCO. Details are provided in Appendix B.



(a) Depth degradation (b) P=32 training curve (c) Parameter stashing (d) Parameter recovery

(e) PipeOptim (f) SpecTrain (g) LWP (h) FRESCO

Figure 4: Main results on 95M language-model pre-training. (a) FRESCO remains robust performance under deeper pipeline compared to other baselines, with (b) prominent optimization performance at $P = 32$. (c–h) training curves across pipeline depths similarly demonstrates robustness of FRESCO.

5 Evaluation

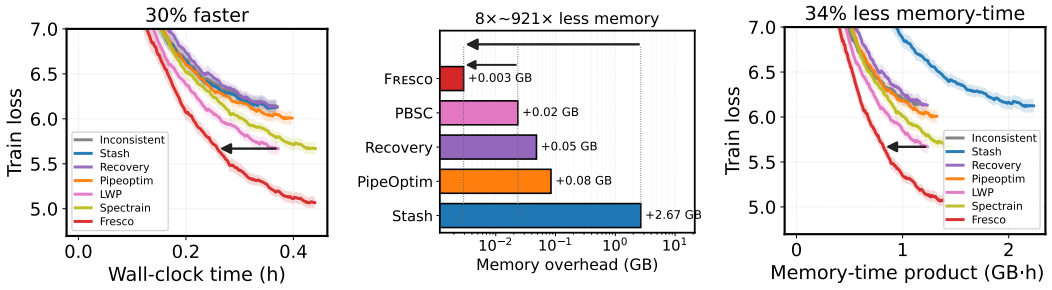
We compare FRESCO with a parameter-inconsistent baseline and five representative asynchronous pipeline-parallel strategies: parameter stashing [32], parameter recovery [47], and three parameter-speculation baselines, SpecTrain, PipeOptim, and LWP [5, 19, 27], which are described in detail in Appendix A.1. Following prior study[1], we evaluate all methods on next-token prediction by pre-training a randomly initialized 95M-parameter NanoGPT decoder-only Transformer [24]. The model is partitioned across $P = 1$ to 32 pipeline stages and trained on OpenWebText (OWT) [17] with a micro-batch size of 8 and a sequence length of 512 for 20k iterations. Additional experimental details are presented in Appendix C. The code to reproduce the results of the paper is provided in `https://github.com/LOG-postech/fresco`.

5.1 Robust optimization under deep pipelines

We evaluate the robustness of FRESCO under increasingly severe inconsistency induced by deep asynchronous pipelining. Specifically, we measure how well each method preserves optimization speed relative to the shallow-pipeline reference as the number of pipeline stages increases up to $P = 32$. Figure 4a reports the final training loss as the number of pipeline stages increases. Most asynchronous baselines degrade sharply with larger P, indicating that their consistency mechanisms become increasingly insufficient as more micro-batches are simultaneously in flight. By contrast, FRESCO maintains a substantially lower final loss under deep pipelines. As shown in Figure 4b, at $P = 32$, FRESCO converges substantially faster than the strongest baseline, LWP, reaching the same training loss with 43.2% fewer iterations.

Additionally, the remaining panels of Figure 4 provide the full training curves across pipeline depths for each method. We report slowdown as the relative number of iterations required to reach a fixed target loss compared with the shallow-pipeline reference. Fully inconsistent asynchronous training, parameter stashing, parameter recovery, and PipeOptim all show slowdowns of approximately 2.80–$2.97\times$, while SpecTrain and LWP still slow down by $2.40\times$ because they cannot perfectly predict the future trajectory. In contrast, FRESCO reduces the slowdown to $1.69\times$. Thus, even though FRESCO uses the current updated parameters rather than reconstructing or predicting a matched full parameter state, subspace-level preservation provides stronger robustness under deep asynchrony.

These results suggest that FRESCO establishes a more reliable strategy than prior baselines, supporting forward-equivalent update projection as effective in preserving optimization progress, avoiding both the stale-gradient limitation of parameter stashing and recovery and the instability of prediction-based



(a) Loss vs. wall-clock time (b) Memory overhead (c) Loss vs. memory-time

Figure 5: Compute efficiency of FRESCO. FRESCO improves both runtime and memory efficiency under deep pipelining: it reaches the target loss about (a) 30% faster in wall-clock time, uses (b) from $8\times$ up to $921\times$ less memory overhead, and achieves a (c) 34% lower memory-time product. These gains come from storing compact activation-subspace summaries rather than delayed full-parameter copies.

methods under uncontrolled asynchronous updates to forward features that pending backward signals still depend on.

5.2 Compute efficiency

We next evaluate the compute efficiency of FRESCO in terms of wall-clock time, memory overhead, and their combined memory-time product on $P = 32$. Figure 5a shows that FRESCO reaches the same target loss about 30% faster than the strongest baseline in wall-clock time, despite the additional computation required for subspace estimation, basis merging, and update projection. Also, the total memory overhead added by each asynchronous pipeline methods in Figure 5b show a complementary advantage. This is due to all baselines rely on parameter-sized auxiliary buffers that is much larger than what FRESCO rely on, with parameter stashing additionally scaling with the number of pipeline stages by storing full parameter versions, reaching about 2.67 GB in our setting. In contrast, FRESCO stores only compact spectral summaries of in-flight activation subspaces, requiring about 3.0 MB of additional memory, corresponding to a $921\times$ reduction over stashing. This also translates into the end-to-end memory-time view in Figure 5c, where the memory-time product measures the cumulative memory footprint required to reach a target loss. By this metric, FRESCO achieves the same loss with a 34% lower memory-time product, showing that its gains do not merely shift cost from memory to additional training time.

6 Discussion

6.1 Protection capacity

Any asynchronous pipeline-parallel method faces a finite consistency budget: as pipeline depth grows, more in-flight micro-batches must be made compatible with later parameter states, increasing either staleness, prediction error, memory, or update restriction. Unlike prior asynchronous pipeline-parallel strategies, FRESCO provides explicit insight into how this capacity scales with pipeline depth: the protected rank should remain well below the layer input dimension, otherwise the feasible update space becomes too small. As a rough estimate, if DeepSeek-V3, with its 61 Transformer blocks and smallest capacity being the expert intermediate width 2048, is considered under its 4K pre-training sequences [29], this would imply that even one delayed batch-size-one micro-batch contains 4096 token features, exceeding the 2048-dimensional input of restrictive expert projections. However, optimistically, recent practical measurements indicate that transformer activation spectra concentrate most of their mass in the top singular directions, fitting $\lambda_j \propto j^{-2\rho}$ with $2\rho \approx 1.2$–1.3 [13], under which the top 64 directions in 7168 dimensions capture about 72%–80% of the spectral mass. Thus, while raw sequence length gives a pessimistic upper bound on the consistency burden, these empirical findings suggest that the effective burden may be much smaller in practice if spectral concentration generalizes, giving reason for optimism about scaling FRESCO and motivating the relaxed protection schemes proposed in Section 3.2.

6.2 Trade-off between protection and progress

Complete protection is likely not always beneficial, even when it is possible. Indeed, the underlying nature of asynchronous pipelining is not merely that each update must protect an increasing number of in-flight micro-batches, but that this protection is inherently in tension with the update's own contribution to optimization progress. Thus, practical schemes that expose dials for this balance, such as relaxed versions of FRESCO, may be important for identifying configurations that maximize performance within a fixed training time. Another possible remedy is to improve the trade-off itself by strengthening the update or derivative oracle, as recent directions such as Nesterov-style updates or basis-rotation methods suggest that stronger optimization techniques may be naturally more robust to inconsistency, perhaps because their local approximation remains useful over a larger region [9, 1, 23]. Together, these observations suggest that future asynchronous pipeline training methods should not treat consistency preservation and optimization as separate components, but should instead co-design the projected update rule, the degree of protection, and the optimizer dynamics, especially in regimes where deeper pipelines make update interference increasingly severe.

6.3 Extensions

One potential extension concerns the update components discarded by FRESCO. Although these components interfere with in-flight forward computations, they may still contain useful optimization information. Rather than discarding them permanently, one could accumulate and reapply them once the corresponding protection constraints expire, resembling error-feedback mechanisms in efficient-state optimizers and distributed optimization [42, 38]. However, these delayed updates may themselves become stale, making their effective reuse a nontrivial problem.

Another extension is to study how FRESCO interacts with hybrid parallelism in large-scale training, including data, tensor, sequence/context, and recent paradigms such as SWARM parallelism [41, 26, 50, 39]. Subspace Networks [36] are particularly relevant, as they argue that training can proceed with low-rank micro-batch forward and error signals, suggesting a naturally favorable setting for FRESCO. A larger-scale stress test across model sizes, pipeline depths, and hybrid-parallel configurations is therefore an important next step, though beyond our available compute.

Finally, a more comprehensive convergence analysis would need to account for the coupled dynamics between parameter changes, the resulting asynchronous gradient errors, and the effects of different update-control mechanisms on subsequent optimization. Characterizing these feedback interactions through a dynamical-systems perspective remains an interesting direction for future work.

7 Conclusion

We proposed a novel and principled *forward-equivalence condition* that directly controls the asynchronous updates, leading us to designing FRESCO based on constrained optimization that demonstrates consistently strong performance and efficiency under deep pipelines compared to existing representative baselines. We believe this shift from parameter-level consistency provides a promising foundation for future high-utilization training systems for stable and efficient training of larger models and more resource-limited distributed environments.

Acknowledgment

This work was partly supported by the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (RS-2019-II191906, Artificial Intelligence Graduate School Program (POSTECH), RS-2026-25507427, Development of Efficient Architectures and Training Techniques for High-Performance Lightweight AI Models), the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (Grant No. RS-2026-25612350), Project Title: Development of an End-to-End Design and Validation Framework for Mathematically Principled Next-Generation AI Architectures and Learning Algorithms), the Korea Basic Science Institute(National research Facilities and Equipment Center) grant funded by the Korea government(MSIT) (RS-2026-25500419).

References

[1] Thalaiyasingam Ajanthan, Sameera Ramasinghe, Yan Zuo, Gil Avraham, and Alexander Long. Nesterov method for asynchronous pipeline parallel optimization. *ICML*, 2025.

[2] B. R. Badrinath and Krithi Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[3] Gerard M Baudet. Asynchronous iterative methods for multiprocessors. *Journal of the ACM (JACM)*, 25(2):226–244, 1978.

[4] Daniel Chazan and Willard Miranker. Chaotic relaxation. *Linear algebra and its applications*, 2(2):199–222, 1969.

[5] Chi-Chung Chen, Chia-Lin Yang, and Hsiang-Yun Cheng. Efficient and robust parallel dnn training through model parallelism on multi-gpu platform. *arXiv preprint arXiv:1809.02839*, 2018.

[6] Huiqiang Chen, Tianqing Zhu, et al. Machine unlearning via null space calibration. In *IJCAI*, 2024.

[7] Zihao Chen, Chen Xu, Weining Qian, and Aoying Zhou. Elastic averaging for efficient pipelined dnn training. In *PPoPP*, 2023.

[8] De Cheng, Yue Lu, Lingfeng He, Shizhou Zhang, Xi Yang, Nannan Wang, and Xinbo Gao. Mamba-CL: Optimizing selective state space model in null space for continual learning. *arXiv preprint arXiv:2411.15469*, 2024.

[9] Timothy Dozat. Incorporating nesterov momentum into adam. *ICLR Workshop*, 2016.

[10] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[11] Junfeng Fang, Houcheng Jiang, Kun Wang, Yunshan Ma, Jie Shi, Xiang Wang, Xiangnan He, and Tat-Seng Chua. Alphaedit: Null-space constrained knowledge editing for language models. In *ICLR*, 2025.

[12] Mehrdad Farajtabar, Navid Azizan, Alex Mott, and Ang Li. Orthogonal gradient descent for continual learning. In *AISTATS*, 2020.

[13] Damien Ferbach, Courtney Paquette, Gauthier Gidel, Katie Everett, and Elliot Paquette. Logarithmic-time schedules for scaling language models with momentum. *arXiv preprint arXiv:2602.05298*, 2026.

[14] Hamid Reza Feyzmahdavian and Mikael Johansson. Asynchronous iterations in optimization: New sequence results and sharper algorithmic guarantees. *arXiv preprint arXiv:2109.04522*, 2023. doi: 10.48550/arXiv.2109.04522.

[15] Andreas Frommer and Daniel B. Szyld. On asynchronous iterations. *Journal of Computational and Applied Mathematics*, 123(1–2):201–216, 2000. doi: 10.1016/S0377-0427(00)00409-X.

[16] Alexander L Gaunt, Matthew A Johnson, Maik Riechert, Daniel Tarlow, Ryota Tomioka, Dimitrios Vytiniotis, and Sam Webster. Ampnet: Asynchronous model-parallel training for dynamic neural networks. *arXiv preprint arXiv:1705.09786*, 2017.

[17] Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[18] Lei Guan, Wotao Yin, Dongsheng Li, and Xicheng Lu. Xpipe: Efficient pipeline model parallelism for multi-gpu dnn training. *arXiv preprint arXiv:1911.04610*, 2019.

[19] Lei Guan, Dongsheng Li, Yongle Chen, Jiye Liang, Wenjian Wang, and Xicheng Lu. Pipeoptim: Ensuring effective 1f1b schedule with optimizer-dependent weight prediction. *IEEE Transactions on Knowledge and Data Engineering*, 2025.

[20] Tuan Hoang, Santu Rana, Sunil Gupta, and Svetha Venkatesh. Learn to unlearn for deep neural networks: Minimizing unlearning interference with gradient projection. In *WACV*, 2024.

[21] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *NeurIPS*, 2019.

[22] Zhouyuan Huo, Bin Gu, Heng Huang, et al. Decoupled parallel backpropagation with convergence guarantee. In *ICML*, 2018.

[23] Hyunji Jung, Sungbin Shin, and Namhoon Lee. Mitigating staleness in asynchronous pipeline parallelism via basis rotation. *arXiv preprint arXiv:2602.03515*, 2026.

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

[25] Boao Kong, Hengrui Zhang, and Kun Yuan. On the convergence of stochastic gradient descent with perturbed forward-backward passes. *arXiv*, 2026.

[26] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *MLSys*, 2023.

[27] Atli Kosson, Vitaliy Chiley, Abhinav Venigalla, Joel Hestness, and Urs Koster. Pipelined backpropagation at scale: training large models without batches. *MLSys*, 2021.

[28] Sen Lin, Li Yang, Deliang Fan, and Junshan Zhang. TRGP: Trust region gradient projection for continual learning. *arXiv preprint arXiv:2202.02931*, 2022.

[29] 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.

[30] Bo Liu, Xingchao Liu, Xiaojie Jin, Peter Stone, and Qiang Liu. Conflict-averse gradient descent for multi-task learning. In *NeurIPS*, 2021.

[31] Yue Lu et al. Visual prompt tuning in null space for continual learning. *NeurIPS*, 2024.

[32] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: Generalized pipeline parallelism for dnn training. *SOSP*, 2019.

[33] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. *ICML*, 2021.

[34] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. Memory-efficient pipeline-parallel dnn training. In *ICML*, 2021.

[35] A Petrowski, G Dreyfus, and C Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 1993.

[36] Sameera Ramasinghe, Thalaiyasingam Ajanthan, Gil Avraham, Yan Zuo, and Alexander Long. Subspace networks: Scaling decentralized training with communication-efficient model parallelism. *NeurIPS*, 2025.

[37] Rodolfo F. Resende, Divyakant Agrawal, and Amr El Abbadi. Semantic locking in object-oriented database systems. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 388–402, 1994.

[38] Thomas Robert, Mher Safaryan, Ionut-Vlad Modoranu, and Dan Alistarh. Ldadam: Adaptive optimization from low-dimensional gradient statistics. *ICLR*, 2025.

[39] Max Ryabinin, Tim Dettmers, Michael Diskin, and Alexander Borzunov. Swarm parallelism: Training large models can be surprisingly communication-efficient. *ICML*, 2023.

[40] Gobinda Saha, Isha Garg, and Kaushik Roy. Gradient projection memory for continual learning. In *ICLR*, 2021.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[42] Sebastian U Stich and Sai Praneeth Karimireddy. The error-feedback framework: Better rates for sgd with delayed gradients and compressed communication. *JMLR*, 2020.

[43] Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[44] Shipeng Wang, Xiaorong Li, Jian Sun, and Zongben Xu. Training networks in null space of feature covariance for continual learning. In *CVPR*, 2021.

[45] William E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.

[46] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–283, 1989.

[47] Bowen Yang, Jian Zhang, Jonathan Li, Christopher Ré, Christopher Aberger, and Christopher De Sa. Pipemare: Asynchronous pipeline parallel dnn training. *MLSys*, 2021.

[48] PengCheng Yang, Xiaoming Zhang, Wenpeng Zhang, Ming Yang, and Hong Wei. Group-based interleaved pipeline parallelism for large-scale dnn training. In *ICLR*, 2022.

[49] Tianhe Yu, Saurabh Kumar, Abhishek Gupta, Sergey Levine, Karol Hausman, and Chelsea Finn. Gradient surgery for multi-task learning. In *NeurIPS*, 2020.

[50] Fanlong Zeng, Wensheng Gan, Yongheng Wang, and Philip S Yu. Distributed training of large language models: A survey. *Natural Language Processing Journal*, 2025.

[51] Guanxiong Zeng, Yang Chen, Bo Cui, and Shan Yu. Continual learning of context-dependent processing in neural networks. *Nature Machine Intelligence*, 2019.

[52] Jiayun Zhang, Shuheng Li, Haiyu Huang, Xiaofan Yu, Rajesh K. Gupta, and Jingbo Shang. Orthogonal calibration for asynchronous federated learning. *arXiv preprint arXiv:2502.15940*, 2025.

[53] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. *ICML*, 2017.

A Related works

A.1 Asynchronous pipeline parallelism methods

Retrospective consistency and staleness One dominant strategy is to compute gradients using the forward-time parameters to ensure consistency between the forward and backward passes [32, 34, 48, 47]. PipeDream [32] introduced *parameter stashing*, which keeps a snapshot of the forward-time parameters for each micro-batch and uses it during the backward pass, analogous to process forking. However, this approach incurs *significant memory overhead*; in the worst case, the first pipeline stage must store parameters equivalent to the entire model. PipeMare [47] addresses this limitation by maintaining an exponential moving average of gradients and applying a first-order Taylor approximation of the optimization trajectory to recover the parameters. Although these approaches produce gradients consistent with the forward-time parameters, they remain stale relative to the latest parameters.

Prospective consistency and forecast error Another line of work predicts the parameters to be used at backward time instead of storing forward-time copies, reducing memory overhead at the cost of prediction error. [5, 18] SpecTrain [5] performs lookahead-style parameter prediction using momentum-smoothed gradients to forecast future parameters at forward time. XPipe [18] refines this by incorporating optimizer-aware update directions (e.g., Adam moments) to improve prediction accuracy and robustness. DC-ASGD [53] instead compensates for delay at the gradient level by applying a first-order Taylor correction, approximating gradients at the current parameters. Similarily to Pipemare, reliance on local approximations to extrapolate non-local parameter trajectories would depend on sufficiently smooth optimization trajectories (often induced by momentum [1]) and may degrade under noisy or rapidly changing gradients.

Restricting optimization progress Another line of work regularizes updates to limit the deviation between forward- and backward-time parameters, reducing inconsistency without explicit prediction or stashing. AvgPipe [7] applies elastic averaging to keep local parameters close to a shared reference, effectively constraining drift across pipeline stages. PipeMare [47] incorporates delay-aware learning rate rescheduling to dampen updates under larger staleness. Momentum correction [1] can be viewed as implicitly smoothing the optimization trajectory, reducing abrupt deviations. However, these methods restrict update dynamics globally, which may limit optimization efficiency and do not directly resolve parameter inconsistency.

A.2 Connection to asynchronous and semantic concurrency control

Our work is conceptually connected to asynchronous numerical computation, which studies alternatives to conventional iterative methods that proceed without global synchronization under delayed or partially inconsistent updates [4, 3, 15, 14]. It is also distantly related to semantic concurrency control, where conflicts are defined by the abstract effects of operations rather than by low-level read–write overlap [45, 46, 2, 37]. We draw only on the shared abstraction principle: synchronization need not always be imposed on the full memory representation when the semantics of the computation identify a smaller conflicting component. Our method can be seen as a domain-specific instantiation of this principle in pipeline-parallel continuous optimization: consistency is enforced on the semantically relevant effect of an update, rather than on its full parameter representation.

A.3 Connection to gradient projection methods in continual and multi-task learning

Projecting parameter updates onto the orthogonal complement of a protected feature subspace recurs across several areas of literature. In *continual learning*, OWM [51], OGD [12], GPM [40], and Adam-NSCL [44] constrain new-task updates to the null space of features from previous tasks, with later work relaxing this constraint or extending it to transformers and state-space models [28, 31, 8]. The same construction appears in *knowledge editing* [11], *multi-task gradient surgery* [49, 30], and *machine unlearning* [6, 20]. The closest precedent is OrthoFL [52], which projects asynchronous federated-learning drift onto a subspace orthogonal to stale client updates. In all of the above, the protected set is essentially stationary—fixed before the update—so the projector can be precomputed once and reused. In our setting, the protected set is the in-flight micro-batches, which turns over

every step; the projector must be rebuilt on the critical path under the per-stage budget of pipeline parallelism.

B Proof of Theorem 4.1

We extend the perturbed forward–backward SGD analysis of Kong et al. [25] to a simplified model of FRESCO. We first bound the perturbations induced by asynchronous parameter changes and then derive the convergence rate under soft projection.

B.1 Model and Assumptions

Consider a pipeline of P stages with linear parameterized operators $f_i(\boldsymbol{w}, \boldsymbol{s}) = W_i \boldsymbol{s}$. For an in-flight micro-batch, let $W_{i,t}^{\mathrm{f}}$, $W_{i,t}^{\mathrm{b}}$, and $W_{i,t}^{\mathrm{c}}$ denote the parameters used during forward propagation, backward propagation, and gradient application, respectively. The corresponding forward and backward perturbations are

$$\begin{aligned} \delta_{i,t}^{\mathrm{f}} &:= (W_{i,t}^{\mathrm{f}} - W_{i,t}^{\mathrm{c}})\boldsymbol{s}_{i,t}, \\ \delta_{i,t}^{\mathrm{b}} &:= (W_{i,t}^{\mathrm{b}} - W_{i,t}^{\mathrm{c}})^{\top}\boldsymbol{e}_{+,i,t}. \end{aligned} \tag{19}$$

These quantities measure the local computational discrepancies relative to the current parameters, holding the input feature and incoming error signal fixed.

At update event u, let $\bar{S}_{i,u}$ collect the protected in-flight features, with compact SVD $\bar{S}_{i,u} = U_{i,u}\Sigma_{i,u}Q_{i,u}^{\top}$. For analysis, we consider the soft-projection update

$$\Delta W_{i,u}' = \Delta W_{i,u} A_{i,u}, \qquad A_{i,u} := (I + \lambda_{i,u} U_{i,u,k} \Sigma_{i,u,k}^2 U_{i,u,k}^{\top})^{-1}. \tag{20}$$

Here $\Delta W_{i,u} = -\eta G_{i,u}$ is the proposed update, with $G_{i,u}$ denoting the matrix representation of the asynchronously computed stage gradient.

The following assumptions are used.

Assumption B.1 (Smoothness)**.** The objective $\mathcal{L}$ is bounded below by $\mathcal{L}_\star$ and has an L-Lipschitz gradient:

$$\|\nabla\mathcal{L}(\boldsymbol{w}) - \nabla\mathcal{L}(\boldsymbol{w}')\| \le L\|\boldsymbol{w} - \boldsymbol{w}'\|. \tag{21}$$

Assumption B.2 (Stochastic gradients)**.** Let $\mathcal{F}_t$ denote the information available before sampling the ideal stochastic gradient, including the current parameters and update operator $\mathcal{A}_t$. The unperturbed gradient $\boldsymbol{g}_t$ satisfies

$$\mathbb{E}[\boldsymbol{g}_t \mid \mathcal{F}_t] = \nabla\mathcal{L}(\boldsymbol{w}_t), \tag{22}$$

and

$$\mathbb{E}[\|\boldsymbol{g}_t - \nabla\mathcal{L}(\boldsymbol{w}_t)\|^2 \mid \mathcal{F}_t] \le \sigma^2. \tag{23}$$

Assumption B.3 (Operator smoothness)**.** For each operator f_i, let $D_1 f_i$ and $D_2 f_i$ denote its input and parameter Jacobians. There exist finite constants $C_1, C_2, L_f, L_1, L_2 > 0$ such that, for all computational states $z = (\boldsymbol{s}, \boldsymbol{w}_i)$ and $z' = (\boldsymbol{s}', \boldsymbol{w}_i')$ under consideration,

$$\|D_j f_i(z)\|_{\mathrm{op}} \le C_1, \qquad j \in \{1, 2\}, \tag{24}$$

$$\|D_{21}^2 f_i(z)\|_{\mathrm{op}} \le C_2, \tag{25}$$

$$\|f_i(\boldsymbol{w}_i, \boldsymbol{s}) - f_i(\boldsymbol{w}_i, \boldsymbol{s}')\| \le L_f\|\boldsymbol{s} - \boldsymbol{s}'\|, \tag{26}$$

and

$$\begin{aligned} \|Df_i(z) - Df_i(z')\|_F &\le L_1\|z - z'\|, \\ \|D^2 f_i(z) - D^2 f_i(z')\|_F &\le L_2\|z - z'\|. \end{aligned} \tag{27}$$

These conditions apply to the differentiable computational graph, including the terminal loss operator.

Assumption B.4 (Bounded computational signals)**.** There exist finite constants $G_i, E_i, M_i > 0$ such that, almost surely,

$$\|G_{i,t}\|_F \le G_i, \qquad \|\boldsymbol{e}_{+,i,t}\| \le E_i, \qquad \|\bar{S}_{i,t}\|_2 \le M_i. \tag{28}$$

Assumptions B.1–B.3 follow the framework of Kong et al. [25]. The stochastic-gradient condition is strengthened to condition on the update operator, which is required for the projected descent argument and is not automatic when the projection depends on the gradient sample. Assumption B.4 supplies uniform bounds on the quantities involved in asynchronous updates.

We consider a fixed pipeline schedule for which the number of intervening stage updates is bounded. Let $\mathcal{H}_{i,t}$ and $\mathcal{J}_{i,t}$ denote the updates separating the forward and backward parameter versions from the current version. The execution model determines finite constants $\tau_i^{\mathrm{f}}, \tau_i^{\mathrm{b}}$ satisfying

$$|\mathcal{H}_{i,t}| \le \tau_i^{\mathrm{f}}, \qquad |\mathcal{J}_{i,t}| \le \tau_i^{\mathrm{b}}. \tag{29}$$

For the convergence analysis, we use a logical-update model that collects the stage-wise soft transformations into a block-diagonal operator $\mathcal{A}_t$:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \eta \mathcal{A}_t \widetilde{\boldsymbol{g}}_t. \tag{30}$$

This model retains the gradient perturbations induced by the pipeline while abstracting away the exact ordering of physical stage updates.

For $0 \le \lambda_{i,t} \le \Lambda < \infty$, the eigenvalues of $A_{i,t}$ lie in $[(1+\Lambda M^2)^{-1}, 1]$, where $M := \max_i M_i$. Hence

$$aI \preceq \mathcal{A}_t \preceq I, \qquad a := \frac{1}{1+\Lambda M^2} > 0. \tag{31}$$

This bound follows from the soft-projection definition.

B.2 Perturbation Bound and Convergence

We first characterize the residual forward perturbation. By the SVD of $\bar{S}_{i,u}$, Equation (20) gives

$$\begin{aligned} R_{i,u}^2 &:= \|\Delta W'_{i,u} \bar{S}_{i,u}\|_F^2 \\ &= \sum_{j=1}^{k} \frac{\sigma_{i,u,j}^2 \|\Delta W_{i,u} u_{i,u,j}\|^2}{(1+\lambda_{i,u}\sigma_{i,u,j}^2)^2} \\ &\quad + \sum_{j=k+1}^{r_{i,u}} \sigma_{i,u,j}^2 \|\Delta W_{i,u} u_{i,u,j}\|^2. \end{aligned} \tag{32}$$

Define

$$s_{i,u} := \max\left\{ \max_{j \le k} \frac{\sigma_{i,u,j}}{1+\lambda_{i,u}\sigma_{i,u,j}^2},\ \sigma_{i,u,k+1} \right\}, \tag{33}$$

with $\sigma_{i,u,r_{i,u}+1} = 0$ and empty maxima equal to zero. Then

$$R_{i,u} \le s_{i,u} \|\Delta W_{i,u}\|_F \le \eta G_i s_{i,u}. \tag{34}$$

Let $s := \sup_{i,u} s_{i,u} \le M$. For hard rank-$k$ projection, obtained as $\lambda \to \infty$, the first sum in Equation (32) vanishes. In particular, protecting the complete activation subspace gives $R_{i,u} = 0$.

Lemma B.5 (Asynchronous gradient error)**.** *Under Assumptions B.3 and B.4 and the bounded-delay execution model, the gradient discrepancy $\mathbf{r}_t := \widetilde{\boldsymbol{g}}_t - \boldsymbol{g}_t$ satisfies*

$$\mathbb{E}\|\mathbf{r}_t\|^2 \le \eta^2 (K_{\mathrm{f}} s^2 + K_{\mathrm{b}}), \tag{35}$$

where K_{f} and K_{b} depend on the computational graph, update delays, and bounds on computational signals.

Proof. The difference between the forward and current parameter states is the sum of intervening updates. Since each such update protects the feature of the affected in-flight micro-batch, Equation (34) gives

$$\begin{aligned} \|\delta_{i,t}^{\mathrm{f}}\| &\le \sum_{u \in \mathcal{H}_{i,t}} R_{i,u} \\ &\le \eta \tau_i^{\mathrm{f}} G_i s. \end{aligned} \tag{36}$$

Similarly, since $\|A_{i,u}\|_2 \le 1$,

$$\begin{aligned}\|\delta^{\mathrm{b}}_{i,t}\| &\le \|\boldsymbol{e}_{+,i,t}\| \sum_{u\in\mathcal{J}_{i,t}} \|\Delta W'_{i,u}\|_2 \\ &\le \eta\tau^{\mathrm{b}}_i G_i E_i.\end{aligned} \tag{37}$$

By the perturbation-propagation result of Kong et al. [25, Theorem 1], there exist nonnegative coefficients $C^{\mathrm{f}}_i, C^{\mathrm{b}}_i$ such that

$$\begin{aligned}\mathbb{E}\|\mathbf{r}_t\|^2 \le \sum_i C^{\mathrm{f}}_i \mathbb{E}\|\delta^{\mathrm{f}}_{i,t}\|^2 \\ + \sum_i C^{\mathrm{b}}_i \mathbb{E}\|\delta^{\mathrm{b}}_{i,t}\|^2.\end{aligned} \tag{38}$$

Substituting Equations (36) and (37) proves the result with

$$\begin{aligned}K_{\mathrm{f}} &:= \sum_i C^{\mathrm{f}}_i (\tau^{\mathrm{f}}_i G_i)^2, \\ K_{\mathrm{b}} &:= \sum_i C^{\mathrm{b}}_i (\tau^{\mathrm{b}}_i G_i E_i)^2.\end{aligned} \tag{39}$$

□

The lemma bounds the squared gradient error by $O(\eta^2)$, with its forward component depending on the residual spectral sensitivity s.

Proof of Theorem 4.1. Write

$$\widetilde{\boldsymbol{g}}_t = \nabla\mathcal{L}(\boldsymbol{w}_t) + \boldsymbol{\xi}_t + \mathbf{r}_t, \tag{40}$$

where $\boldsymbol{\xi}_t := \boldsymbol{g}_t - \nabla\mathcal{L}(\boldsymbol{w}_t)$ is the unperturbed stochastic-gradient noise.

By L-smoothness and Equation (30),

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}_{t+1}) \le \mathcal{L}(\boldsymbol{w}_t) - \eta\langle\nabla\mathcal{L}(\boldsymbol{w}_t), \mathcal{A}_t\widetilde{\boldsymbol{g}}_t\rangle \\ + \frac{L\eta^2}{2}\|\mathcal{A}_t\widetilde{\boldsymbol{g}}_t\|^2.\end{aligned} \tag{41}$$

Let $\mathbf{h}_t := \nabla\mathcal{L}(\boldsymbol{w}_t)$. By conditional unbiasedness,

$$\mathbb{E}[\langle\mathbf{h}_t, \mathcal{A}_t\boldsymbol{\xi}_t\rangle \mid \mathcal{F}_t] = 0. \tag{42}$$

Furthermore, Young's inequality gives

$$-\langle\mathbf{h}_t, \mathcal{A}_t\mathbf{r}_t\rangle \le \frac{1}{4}\|\mathcal{A}_t^{1/2}\mathbf{h}_t\|^2 + \|\mathcal{A}_t^{1/2}\mathbf{r}_t\|^2, \tag{43}$$

and

$$\begin{aligned}\|\mathcal{A}_t\widetilde{\boldsymbol{g}}_t\|^2 \le 2\|\mathcal{A}_t\mathbf{h}_t\|^2 + 4\|\mathcal{A}_t\boldsymbol{\xi}_t\|^2 \\ + 4\|\mathcal{A}_t\mathbf{r}_t\|^2.\end{aligned} \tag{44}$$

Using $aI \preceq \mathcal{A}_t \preceq I$ and $\eta \le 1/(4L)$, taking expectations in Equation (41) yields

$$\begin{aligned}\frac{a\eta}{2}\mathbb{E}\|\nabla\mathcal{L}(\boldsymbol{w}_t)\|^2 \le \mathbb{E}[\mathcal{L}(\boldsymbol{w}_t) - \mathcal{L}(\boldsymbol{w}_{t+1})] \\ + 2L\eta^2\sigma^2 + \frac{3\eta}{2}\mathbb{E}\|\mathbf{r}_t\|^2.\end{aligned} \tag{45}$$

Summing over $t = 0, \dots, T-1$, using $\mathcal{L}(\boldsymbol{w}_T) \ge \mathcal{L}_\star$, and applying Lemma B.5, we obtain

$$\begin{aligned}\frac{1}{T}\sum_{t=0}^{T-1}\mathbb{E}\|\nabla\mathcal{L}(\boldsymbol{w}_t)\|^2 \le \frac{2\Delta_{\mathcal{L}}}{a\eta T} + \frac{4L\eta\sigma^2}{a} \\ + \frac{3\eta^2(K_{\mathrm{f}}s^2 + K_{\mathrm{b}})}{a},\end{aligned} \tag{46}$$

where $\Delta_{\mathcal{L}} := \mathcal{L}(\boldsymbol{w}_0) - \mathcal{L}_\star$.

Choosing $\eta = c/\sqrt{T}$ proves Equation (18). □

Table 1: **Common experimental configuration.** Hyperparameters marked as swept are selected separately for each method and pipeline depth.

Configuration	Value
Model	
Architecture	nanoGPT decoder-only Transformer
Embedding dimension d	384
Attention heads a	6
Transformer blocks L	32
Context length s	512
Parameters	$\sim$ 95M
Positional embeddings	Learnable
LM head	Untied
Training	
Dataset	OpenWebText
Micro-batch size b	8
Precision	`FP32`
Optimizer	NAdamW
β_1 (swept)	$\{0.9, 0.99\}$
β_2, ϵ	0.999, 10^{-8}
Weight decay	0.01
Gradient clipping	1.0
Peak LR (swept)	$\{10^{-4}, 3 \times 10^{-4}, 10^{-3}\}$
LR warmup	3,000 steps, linear
LR decay	Cosine to $0.1\times$ peak LR
Training horizon	20,000 steps
Random seed	0
Distributed Setup	
Pipeline depths P	$\{1, 2, 4, 8, 16, 32\}$
GPUs per run	8
Training GPUs	RTX 3090 / RTX A6000
Timing GPUs	$8\times$ RTX 3090 (24 GB)
Backend	`gloo`
Activation recomputation	Disabled

The bound preserves the standard $O(T^{-1/2})$ nonconvex stochastic convergence rate, with an additional $O(T^{-1})$ term depending on the residual forward sensitivity and backward perturbations. It does not imply that stronger projection necessarily improves convergence, since the minimum retained update gain a also decreases with the protection strength. The guarantee concerns the simplified soft-projection model; convergence of the implemented hard projection under the full asynchronous execution dynamics remains outside its scope.

C Experimental Details

C.1 Model and Training Configuration

Our experimental setup generally follows Ajanthan et al. [1]. We use a nanoGPT-style decoder-only Transformer [24] with $L = 32$ Transformer blocks, embedding dimension $d = 384$, $a = 6$ attention heads, and context length $s = 512$, comprising approximately 95M parameters. The model uses learnable positional embeddings and an untied language modeling head. At pipeline depth P, each stage contains L/P Transformer blocks, with the embedding layer assigned to the first stage and the LM head to the last.

We train on OpenWebText for 20,000 optimizer steps with micro-batch size $b = 8$ and `FP32` precision. We use NAdamW with weight decay 0.01, gradient clipping at 1.0, and a 3,000-step linear warmup followed by cosine decay to 0.1 times the peak learning rate. We disable activation recomputation for all methods. The common configuration is summarized in Table 1.

C.2 Hyperparameter Selection

Learning rate and momentum. For each method and pipeline depth $P \in \{1, 2, 4, 8, 16, 32\}$, we sweep the peak learning rate and first-moment coefficient over

$$\eta_{\max} \in \{10^{-4},\, 3 \times 10^{-4},\, 10^{-3}\}, \qquad \beta_1 \in \{0.9,\, 0.99\}. \tag{47}$$

Each configuration is evaluated with a single seed (0). For the main results, we select the configuration with the lowest final training loss, measured by an exponential moving average (EMA) with decay factor 0.97, evaluated at step $19{,}900$.

Ablations over the learning rate, β_1, learning-rate damping, and the protection budget are reported in Appendix D.

Protection budget. We select the protection budget α at the deepest pipeline depth, $P = 32$, over the full $20{,}000$-step training horizon. Specifically, we sweep

$$k \in \{12,\, 24,\, 48,\, 64,\, 96,\, 154,\, 269,\, 384\}, \qquad \alpha = \frac{k}{384}. \tag{48}$$

For each linear layer, the protected rank is $\lceil \alpha \min(d_{\text{out}}, d_{\text{in}}) \rceil$, with bias augmentation adding an input coordinate, so $k = \alpha \cdot 384$ for the 384-dimensional layers ($k + 1$ for the 385-dimensional ones). Each budget is evaluated using its best learning rate and β_1. The results are shown in Figure 7a.

We select $\alpha = 96/384 = 0.25$ based on the lowest evaluation loss. Smaller budgets provide insufficient protection against forward interference, whereas larger budgets increasingly restrict the available update directions. Full-rank protection can reach a lower training loss (4.98 at a peak LR of 10^{-3}) but then generalizes poorly (evaluation loss 6.19), which is why we select budgets by evaluation loss. We use $\alpha = 0.25$ for the pipeline-depth sweep. For wall-clock comparisons, we instead use $\alpha = 48/384 = 0.125$, which provides the best loss at a fixed wall-clock budget.

For Recovery, we similarly sweep the discrepancy-correction coefficient over

$$D \in \{0.01,\, 0.02,\, 0.05,\, 0.1\}, \tag{49}$$

and use $D = 0.02$ throughout.

C.3 Subspace Extraction and Projection

FRESCO uses bias-augmented projection: each input activation matrix gains a constant row of ones, $\bar{X} = [X; \mathbf{1}^\top] \in \mathbb{R}^{d \times n}$ with $d = d_{\text{in}} + 1$, so that the projection also protects the bias term. With the protected rank k and τ in-flight micro-batches, each micro-batch contributes $k_1 = \lceil k/\tau \rceil$ directions.

Subspace extraction. EXTRACTBASIS$(\bar{X}, k_1)$ approximates the leading k_1 left singular vectors of $\bar{X}$ by block power iteration. From an initial block $V_0 \in \mathbb{R}^{d \times k_1}$, we iterate

$$V_{j+1} = \mathrm{NS}_{T_{\text{ns}}}(\bar{X}\bar{X}^\top V_j), \qquad j = 0, \ldots, T_{\text{bp}} - 1, \tag{50}$$

where NS_T normalizes its argument, $C \leftarrow C/\|C\|_F$, and applies T Newton–Schulz steps $C \leftarrow \frac{1}{2}C(3I - C^\top C)$, and return $V = V_{T_{\text{bp}}}$. We use $T_{\text{bp}} = 1$ and $T_{\text{ns}} = 1$, so each micro-batch costs two multiplications by its activations and a few small matrix products.

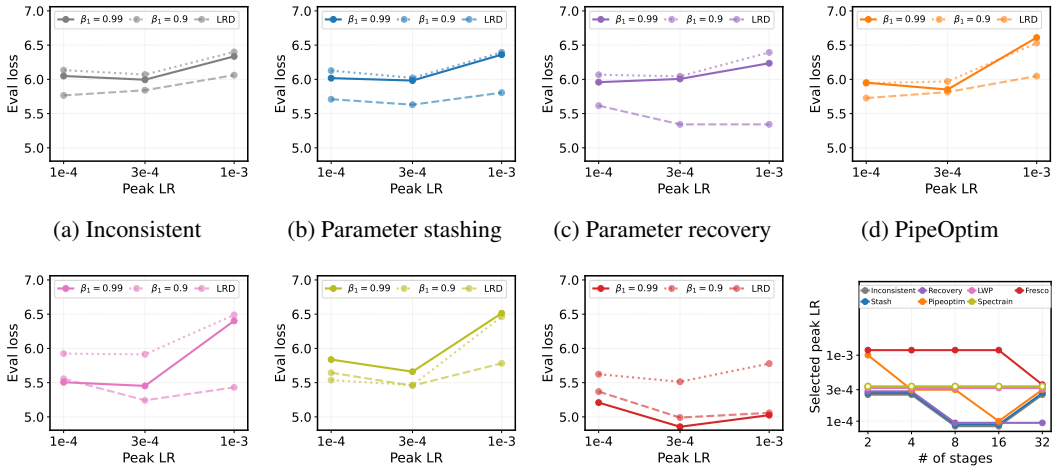
Protected-basis construction. BUILDPROTECTEDBASIS$(\mathcal{Q}, k)$ combines the bases $\mathcal{Q} = \{V^{(1)}, \ldots, V^{(\tau)}\}$ of the in-flight micro-batches into a rank-k orthonormal basis. Their concatenation

$$C = [V^{(1)}, \ldots, V^{(\tau)}] \in \mathbb{R}^{d \times \tau k_1}, \qquad \tau k_1 \geq k, \tag{51}$$

spans up to τk_1 directions. When C has at most k columns, we take $Q = \mathrm{qr}(C)$; otherwise, we keep the dominant k directions of C by block power iteration from an initial block $Q_0 \in \mathbb{R}^{d \times k}$,

$$Q_{j+1} = \mathrm{qr}(CC^\top Q_j), \qquad j = 0, \ldots, T_{\text{c}} - 1, \tag{52}$$

with $T_{\text{c}} = 2$, where qr returns the orthonormal factor of a Householder QR decomposition. The resulting $Q \in \mathbb{R}^{d \times k}$ is used to project the proposed parameter update onto the complement of the protected subspace, $\Delta' = \Delta(I - QQ^\top)$.



(a) Inconsistent (b) Parameter stashing (c) Parameter recovery (d) PipeOptim

(e) LWP (f) SpecTrain (g) FRESCO (h) Selected LR per depth

Figure 6: Learning rate, momentum, and learning-rate damping. (a–g) Evaluation loss over the peak learning rate at $P = 32$ with $\beta_1 = 0.99$ (solid) and $\beta_1 = 0.9$ (dotted), both without LRD, and with LRD at the best β_1 (dashed). (h) The peak LR selected for each method and pipeline depth in the main results (Figure 4); hollow markers denote $\beta_1 = 0.9$, and coinciding lines are offset slightly for visibility.

C.4 Hardware and Pipeline Scheduling

Experiments are performed on a GPU cluster comprising 3 nodes with 8 NVIDIA RTX 3090 (24 GB) GPUs per node, with RTX A6000 GPUs also used for training runs. Each run uses 8 GPUs, and wall-clock measurements are conducted on a single node with 8 RTX 3090 GPUs. Communication uses the `gloo` backend.

The pipeline depth varies over $P \in \{1, 2, 4, 8, 16, 32\}$. When $P > 8$, multiple virtual pipeline stages are assigned to each physical GPU, giving $P/8$ stages per GPU. This preserves the logical pipeline depth, the number of in-flight micro-batches, and the order of forward, backward, and update operations. The resulting wall-clock measurements may include additional context-switching overhead, which is shared across the compared methods.

D Hyperparameter Ablations

We vary the peak learning rate, β_1, and learning-rate damping at $P = 32$, and report the learning rate selected at each depth. Each point is the lowest evaluation loss over β_1 (unless varied) and the method-specific setting (α for FRESCO, D for Recovery); all runs use a single seed.

D.1 Learning rate

Every method except Recovery attains its lowest evaluation loss at a peak LR of 3×10^{-4} (Figure 6), and FRESCO has the lowest loss at every learning rate. FRESCO is also the least sensitive to a larger step size: raising the LR from 3×10^{-4} to 10^{-3} increases its loss by 0.17, compared with 0.23–1.00 for the baselines. The three prediction-based methods degrade the most (PipeOptim 0.68, LWP 0.95, SpecTrain 1.00), consistent with larger updates making their predicted weights less accurate.

Across depths (Figure 6h), Inconsistent, Stash, and Recovery select the smallest LR (10^{-4}) at $P = 8$ and 16 (Recovery also at $P = 32$), and PipeOptim at $P = 16$. FRESCO instead selects the largest LR (10^{-3}) up to $P = 16$ and 3×10^{-4} at $P = 32$, a $10\times$ larger step than these baselines at $P = 8$ and 16. LWP and SpecTrain select 3×10^{-4} at every depth.

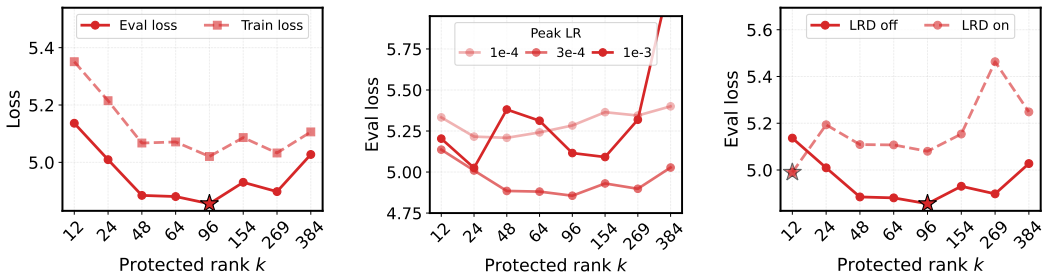
D.2 Momentum

$\beta_1 = 0.99$ gives a lower loss than $\beta_1 = 0.9$ for every method except SpecTrain (5.66 versus 5.47; Figure 6). The coefficient matters most for FRESCO ($5.51 \to 4.86$) and LWP ($5.91 \to 5.45$); for the other baselines the difference is at most 0.09. Across depths, every method that is swept over both values selects $\beta_1 = 0.99$ (Figure 6h). Because the preferred β_1 differs across methods, we tune it per method rather than fixing it.

D.3 Learning-rate damping

We evaluate the staleness-aware learning-rate damping (LRD) of PipeMare [47], which sets the learning rate of a stage with delay τ to $\eta_t \tau^{-p_t}$ with $p_t = 1 - \min(t/K, 1)$, where η_t is the scheduled learning rate at step t; we use $K = 15{,}000$. LRD improves every baseline except SpecTrain, which is essentially unaffected ($5.47 \to 5.46$; Figure 6); the largest gain is for Recovery ($5.96 \to 5.34$), whose discrepancy correction PipeMare proposes together with LRD. In contrast, LRD increases the evaluation loss of FRESCO from 4.86 to 4.99 and shifts its best budget to the smallest one, $\alpha = 12/384$ (Figure 7c). One possible explanation is that LRD and the projection of FRESCO both attenuate the updates that would interfere with in-flight micro-batches, so that combining them over-damps optimization. Even when LRD is enabled for every baseline, FRESCO without LRD remains the best configuration, 0.39 below the strongest damped baseline (LWP, 5.24), although LRD reorders the baselines (e.g., Recovery moves from fourth to second). We therefore disable LRD for all methods in the main comparisons, so that every method uses the same learning-rate schedule.

D.4 Protection budget



(a) Budget at $P = 32$ (b) Budget and LR (c) Budget and LRD

Figure 7: Protection budget of FRESCO over the protected rank k ($\alpha = k/384$). (a) Evaluation loss and training loss (EMA 0.97 at step 19,900) at $P = 32$, each budget at its best peak LR and β_1 by evaluation loss (3×10^{-4} and 0.99 for every budget). (b) One line per peak LR at $P = 32$. (c) LRD off and on at $P = 32$. Stars in (a) and (c) mark the best budget of each line.

At $P = 32$, the evaluation loss is lowest at $\alpha = 96/384$ and stays within 0.08 of it over $\alpha \in [48/384, 269/384]$ (Figure 7a). A peak LR of 3×10^{-4} gives the lowest loss at every budget (Figure 7b), so the budget can be selected at a single learning rate. At 10^{-3}, however, large budgets degrade sharply, up to an evaluation loss of 6.19 at full rank, while at 10^{-4} the loss varies by less than 0.2 across budgets.