Dr. Post-Training 🩺: A Data Regularization Perspective on LLM Post-Training

Pingbang Hu[1]* **Xueshen Liu**[2] **Z. Morley Mao**[2] **Jiaqi W. Ma**[1]*
[1]University of Illinois Urbana–Champaign [2]University of Michigan
{pbb,jiaqima}@illinois.edu {liuxs,zmao}@umich.edu

Abstract

Data selection methods address a critical challenge in LLM post-training: effectively leveraging scarce, high-fidelity target data alongside abundant but imperfectly aligned general training data. In this work, we move beyond the data-selection framing and introduce **Dr. Post-Training** (**D**ata-**R**egularized Post-Training), a novel framework that reconceptualizes general training data as a data-induced regularizer that prevents overfitting to the scarce target objective, rather than serving as a pool for selection. Specifically, our framework proposes that at each training step, construct a feasible set of model update directions using the general training data, and project the model update direction specified by the scarce target data onto that feasible set. Standard training and existing data selection methods arise as special cases with different choices of the data-induced regularizer, and these methods correspond to different points on a bias–variance spectrum with different regularization strength. Building on this view, we propose a family of methods offering a richer design space and more flexible bias–variance tradeoffs. For practical LLM-scale use, we introduce careful system optimizations that realize these methods with minimal overhead. Extensive experiments across SFT, RLHF, and RLVR show that our methods consistently outperform state-of-the-art data selection baselines, and system benchmarks confirm their efficiency.

1 Introduction

Post-training is a critical stage where pretrained large language models (LLMs) are adapted to desired downstream behaviors, including instruction following, dialogue, safety alignment, and domain-specific reasoning. It is typically carried out through supervised fine-tuning (SFT) [Mishra et al., 2022, Muennighoff et al., 2024] or reinforcement learning (RL) variants such as reinforcement learning from human feedback (RLHF) [Christiano et al., 2017, Ouyang et al., 2022] and reinforcement learning with verifiable rewards (RLVR) [Guo et al., 2025]. In many practical post-training settings, however, the data that most directly reflects the downstream objective is scarce: high-quality demonstrations [Ouyang et al., 2022], domain-specific examples [Ling et al., 2025], preference labels [Casper et al., 2023], or task-specific validation data [Perez et al., 2021] are often expensive to obtain. On the other hand, practitioners often have access to a large pool of general training data, such as broad instruction-tuning mixtures [Wang et al., 2023a, Lambert et al., 2024], which are abundant and cheaper but only partially aligned with the target objective [Gururangan et al., 2020, Xia et al., 2024a, He et al., 2024]. This creates a natural tradeoff in post-training: training on the scarce target data is faithful to the downstream objective but statistically noisy, while training on the abundant general data is more stable but may be biased by the misalignment with the target.

This tradeoff has led to a growing literature on data-centric methods for post-training, including heuristic filtering [Bai et al., 2022, Ethayarajh et al., 2022], data mixture optimization [Iyer et al.,

*Corresponding to: Pingbang Hu and Jiaqi W. Ma

Preprint.

2022, Wang et al., 2023a], offline data selection [He et al., 2024, Xia et al., 2024a], and online data selection [Wang et al., 2024b, Hu et al., 2025b]. Despite their algorithmic differences, most existing approaches are operationalized through prioritizing "high-utility" samples from the general training data according to some notion of utility, which makes it natural to frame the problem as *data selection* [Albalak et al., 2024].

However, the data selection perspective obscures a more fundamental question: how to effectively leverage abundant but imperfectly aligned general training data to improve a target objective defined by much scarcer data. Asking how this general dataset should influence the model's updates, rather than just which data to select, opens up a much broader design space.

In this work, we introduce **Dr.** (**D**ata-**R**egularized) Post-Training, a framework that formalizes this idea. At each training step, the target signal specifies the ideal direction for the model's update, while the general training data constrains the update by projecting it onto a feasible set supported by the available training data. From this viewpoint, general training data act as a data-induced *regularizer*: they do not define the target objective itself, but restrict the admissible ways the model can move toward it, preventing overfitting to the scarce target data.

Interestingly, standard training and existing data selection methods arise as special cases of the proposed framework with different choices of the data-induced regularizer, and they can be viewed as different points on a bias–variance spectrum, corresponding to different regularization strengths. Building on this framework, we further propose new methods with *Group-Wise Subset Update* that explicitly tune the strength of this data regularization, enabling more flexible tradeoffs between approximation bias and statistical variance.

Making this broader design space useful in practice requires addressing a second challenge: efficient implementation at LLM scale. Data-regularized updates require comparing training samples against the target signal and then assembling the final update from the selected samples. These operations introduce per-sample dependencies that are absent from standard training, where backpropagation usually computes only the aggregate batch gradient. A naive implementation would either run additional forward–backward passes or retain much more intermediate information than standard training, both of which are undesirable under the memory constraints of modern post-training pipelines.

To overcome this challenge, we conduct careful system-level optimizations to develop an efficient realization of Dr. Post-Training. We first develop a customized tensor lifetime scheduling strategy that selectively retains and releases intermediate quantities in the computation graph to obtain the data-regularized update within one forward–backward pass with minimal memory overhead. We further develop efficient approximate algorithms that significantly mitigate the main runtime bottleneck. We also demonstrate the compatibility between the proposed methods and modern memory-saving techniques such as LoRA finetuning or activation checkpointing.

Empirically, we validate Dr. Post-Training across the three post-training paradigms: SFT, RLHF, and RLVR. Across all settings, our proposed methods consistently outperform both standard training and prior state-of-the-art data-centric baselines, while incurring minimal system overhead.

In summary, our contributions are:

- **Framework.** We introduce **Dr. Post-Training**, a novel post-training framework that employs abundant yet imperfectly aligned general training data as data regularizers, with existing data selection methods as special cases.
- **Method.** Based on the Dr. Post-Training framework, we propose *Group-Wise Subset Update*, which provides a broader design space of data regularizers with more flexible tradeoffs between approximation bias and statistical variance.
- **System.** We show that the proposed methods admit efficient implementations under the memory constraints of modern LLM training.
- **Experiments.** We demonstrate consistent improvements of downstream performance over strong baselines across SFT, RLHF, and RLVR, and conduct extensive system benchmarking to validate the efficiency of our implementation.

2 Related Work

2.1 Data Optimization for LLM Post-Training

Data optimization for LLM post-training aims to improve downstream performance by prioritizing high-utility training examples from large, heterogeneous corpora. We summarize several common families and refer readers to Albalak et al. [2024], Deng et al. [2025] for a comprehensive overview.

Most existing methods approach this as a *data selection* problem, leveraging downstream performance-related signals to determine which training examples the model sees. These span heuristic filtering and quality scoring [Bai et al., 2022, Ethayarajh et al., 2022, Chen et al., 2024, Ivison et al., 2023, Lu et al., 2024, Kung et al., 2023], dataset-level mixture design [Cao et al., 2023, Wei et al., 2022, Iyer et al., 2022, Wang et al., 2023a], and offline example-level selection via influence-style estimators, validation-based scoring, or importance weighting [Xia et al., 2024a, He et al., 2024, Wang et al., 2024a, Koh and Liang, 2017, Pruthi et al., 2020, Ghorbani and Zou, 2019]. Online methods adapt selection during training using per-step signals (e.g., loss- or gradient-based) to dynamically choose or weight examples [Wang et al., 2024b, Jiao et al., 2025, Hu et al., 2025b], better tracking non-stationary utility at the cost of tighter compute constraints. In contrast, our proposed framework takes the *dual* perspective, where the training examples are used to regularize the admissible downstream performance-driven updates, and unlocks a novel and broader design space.

2.2 Memory Saving Techniques in LLM Post-Training

Modern LLM training is typically under stringent computational constraints, especially memory bottlenecks. Extensive research effort has therefore been devoted to parameter-efficient methods and memory-saving scheduling techniques that allow post-training to fit within available hardware budgets.

A common strategy restricts learning to a low-dimensional subspace. Low-Rank Adaptation (LoRA) [Hu et al., 2022] injects trainable low-rank adapters into selected layers while keeping the pretrained parameters frozen, significantly reducing memory and compute requirements [Han et al., 2024, Renduchintala et al., 2024, Sheng et al., 2023, Zhang et al., 2023, Xia et al., 2024b, Wang et al., 2023b, Dettmers et al., 2023]. Complementary to LoRA, Memory-efficient Subspace Optimization (MeSO) methods [Zhao et al., 2024, Muhamed et al., 2024, He et al., 2025] instead compress optimizer states and gradients by projecting them into a low-dimensional representation, applying updates in that space, and mapping them back to the original parameter space for actual parameter updates. We provide further details in Section A.

Complementary to low-dimensional subspace methods, memory-saving techniques restructure how computation is scheduled to further lower peak memory. Activation checkpointing [Chen et al., 2016] discards intermediate activations during the forward pass and recomputes them on-the-fly during the backward pass, trading additional computation for reduced memory footprint. Another technique, called gradient accumulation, is widely adopted in popular large-scale training frameworks [Rasley et al., 2020, von Werra et al., 2020, Sheng et al., 2024], partitions a large batch into micro-batches processed sequentially, accumulating gradients across micro-batches to simulate a larger effective batch size at a fraction of the peak memory cost. Both techniques preserve the optimization trajectory exactly and are widely composed with parameter-efficient methods in practice, making the resulting memory management landscape particularly intricate for methods that introduce additional per-sample computational dependencies beyond standard training.

In this work, we demonstrate that our proposed method integrates seamlessly with these techniques, showing its compatibility with common post-training pipelines (Section 4.4).

3 Dr. Post-Training : A Data Regularization Framework

We now introduce the **Dr. Post-Training** framework, which views general training data *not* as a pool of examples to select from, but as a *data regularizer* for optimizing a target objective. We consider a post-training setup (Section 3.1) with access to two data sources: abundant general training data and limited target data. The framework is formulated through an iterative optimization procedure (Section 3.2) where each step draws a training batch and a target batch from these two sources. The

target batch specifies the objective, while the training batch regularizes the optimization by defining a *feasible set* of parameter update directions. Under this framework, standard training and existing data selection methods arise as special cases through different definitions of the feasible set induced by the training batch (Section 3.3.2). Moreover, the framework suggests a broader design space that explicitly tunes the strength of data regularization, leading to a new family of methods (Section 3.3.3). Next, we show that the methods within this framework lie on a spectrum with a bias–variance tradeoff (Section 3.4). Finally, we conclude this section with a remark on how the proposed framework may generalize to broader training paradigms (Section 3.5).

3.1 Problem Setup

Consider a post-training setup where a practitioner has access to two data distributions: a *general training distribution*, from which abundant training data can be drawn, and a *target distribution*, for which only limited data are available. The target distribution represents the downstream objective of interest, while the general training distribution provides a broader but potentially misaligned source of supervision. Our goal is to understand how to best leverage the abundant general training data to improve performance on the target objective.

This setup arises naturally in several common post-training scenarios: (i) general-purpose SFT, where a curated mixture of diverse tasks defines the training distribution and a specific downstream benchmark serves as the target [Lambert et al., 2024, Grattafiori et al., 2024, Qwen Team, 2025], (ii) domain-adaptive fine-tuning, where a broad general corpus supplements scarce domain-specific data [Gururangan et al., 2020], and (iii) targeted instruction tuning, where representative examples from the target task guide selection from a large instruction corpus [Xia et al., 2024a, He et al., 2024, Wang et al., 2024b].

Notation. We now introduce the necessary notation to formalize this setup. Let $\ell(\theta; z)$ be a differentiable loss function over model parameters $\theta \in \mathbb{R}^d$ and a data sample z. Let the target distribution be $\mathbb{P}_\star$, the corresponding population loss and the gradient are then defined as $\mathcal{L}_\star(\theta) \coloneqq \mathbb{E}_{z\sim\mathbb{P}_\star}[\ell(\theta; z)]$ and $g_\star(\theta) \coloneqq \nabla_\theta \mathcal{L}_\star(\theta) \in \mathbb{R}^d$. Similarly, for the general training distribution $\mathbb{P}_{\mathrm{tr}}$, we define $g_{\mathrm{tr}}(\theta) \coloneqq \nabla_\theta \mathcal{L}_{\mathrm{tr}}(\theta)$ such that $\mathcal{L}_{\mathrm{tr}}(\theta) \coloneqq \mathbb{E}_{z\sim\mathbb{P}_{\mathrm{tr}}}[\ell(\theta; z)]$.

3.2 The Data-Regularization Framework for Post-Training

We now formalize the central asymmetry in our setting: the target data determine *what* objective should be optimized, while the general training data determine *how* the update is allowed to move. Our framework adopts an iterative optimization view in which, at each step, the update direction is chosen to improve the target objective, but only from a set of directions induced by the training batch. In this way, the training data act as a *data regularizer*: they do not specify the target objective itself, but instead constrain the admissible target-driven updates.

3.2.1 Framework

Target-driven one-step update. Modern post-training is carried out by iterative optimization [Wei et al., 2022, Schulman et al., 2017, Ouyang et al., 2022, Shao et al., 2024], where at step t, the model parameters θ_t is updated via

$$\theta_{t+1} = \theta_t - \eta_t u_t,$$

where $\eta_t > 0$ is the step size and $u_t \in \mathbb{R}^d$ is the update direction. Since the target objective is the ultimate quantity of interest, we adopt a one-step target-driven view and choose the update direction to minimize the next-step target loss within a prescribed *feasible set* U_t:

$$u_t = \arg\min_{u\in U_t} \mathcal{L}_\star(\theta_t - \eta_t u). \tag{1}$$

The role of the feasible set is central: it determines which update directions are allowed at step t.

Training-batch-induced feasible set. At each step, we draw two batches:

$$B_t = \{z_i\}_{i=1}^n \overset{\text{i.i.d.}}{\sim} \mathbb{P}_{\mathrm{tr}}, \qquad B_t^\star = \{z_j^\star\}_{j=1}^m \overset{\text{i.i.d.}}{\sim} \mathbb{P}_\star.$$

The training batch B_t is used to construct the feasible set U_t, while the target batch $B_t^\star$ is used to evaluate which direction in U_t best decreases the target loss $\mathcal{L}_\star$.

More concretely, for any set of data B, let $g_B(\theta_t)$ denotes the batch gradient $\frac{1}{|B|}\sum_{z\in B}\nabla_\theta \ell(\theta_t; z)$, and for convenience, we further write

$$g_i(\theta_t) \coloneqq g_{\{z_i\}}(\theta_t) = \nabla_\theta \ell(\theta_t; z_i), \qquad g_j^\star(\theta_t) \coloneqq g_{\{z_j^\star\}}(\theta_t) = \nabla_\theta \ell(\theta_t; z_j^\star)$$

denote the per-sample gradients from the training and target batches, respectively, and define

$$\hat{g}_{\mathrm{tr}}(\theta_t) \coloneqq g_{B_t}(\theta_t) = \frac{1}{n}\sum_{i=1}^{n} g_i(\theta_t), \qquad \hat{g}_\star(\theta_t) \coloneqq g_{B_t^\star}(\theta_t) = \frac{1}{m}\sum_{j=1}^{m} g_j^\star(\theta_t).$$

We fix a step t and omit writing θ_t when it is clear from the context hereafter, e.g., g_i, $g_j^\star$, $\hat{g}_{\mathrm{tr}}$, $\hat{g}_\star$, and also the population gradients $g_\star \coloneqq g_\star(\theta_t)$ and $g_{\mathrm{tr}} \coloneqq g_{\mathrm{tr}}(\theta_t)$.

The key modeling choice in our framework is that U_t depends only on the training batch B_t, typically through the collection of training gradients $\{g_i\}_{i=1}^n$, whereas the target batch $B_t^\star$ enters only through the target objective via information such as $\hat{g}_\star$, as we will soon see. This separation makes explicit the distinct roles of the two data sources: the target batch identifies the desired direction of improvement, while the training batch constrains which directions are admissible.

3.2.2 The Dual Perspective of Data Selection and Data Regularization

The formulation above admits a dual perspective of *data selection* and *data regularization*, as illustrated in Figure 1.

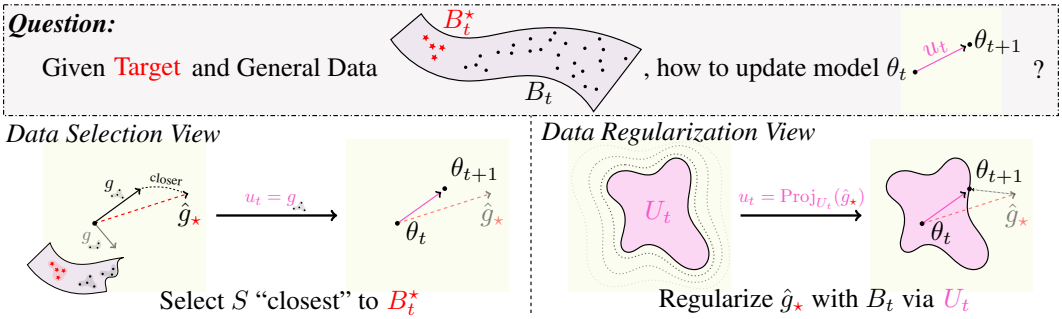

Figure 1: The dual view of data selection and data regularization.

The conventional data selection view. The target objective determines which training examples are most useful, and the update is formed from those selected examples. This is the perspective taken by most existing data selection methods [Xia et al., 2024a, He et al., 2024, Wang et al., 2024b].

The novel data regularization view. The target objective specifies the ideal direction of improvement, while the training batch regularizes this update by restricting it to lie in the feasible set U_t. Without such a regularization, one would attempt to optimize the target objective directly, but this can be statistically unstable when target data are scarce. Constraining the update to directions supported by the training batch can reduce this instability, at the price of introducing approximation bias when the training and target distributions are misaligned.

This viewpoint suggests that the complexity of the feasible set controls a **bias–variance tradeoff**. A smaller feasible set imposes stronger regularization: it stabilizes optimization but may bias the update away from the ideal target direction. A larger feasible set weakens this regularization: it allows more faithful target-driven updates, but may incur higher statistical variance. We formalize this tradeoff in Section 3.4 after introducing some concrete instantiations of the feasible sets below.

3.3 Concrete Instantiations of the Feasible Sets

We next turn the framework into concrete methods. We begin by deriving a tractable approximation of the optimization problem in Eq.(1) (Section 3.3.1). With this approximation, several existing methods

arise as special cases by specifying how the training batch induces the feasible set U_t (Section 3.3.2). We then use this perspective to motivate a broader design space of data regularizer via group-wise decomposition (Section 3.3.3).

3.3.1 A Tractable Approximation of Eq.(1)

We first derive a tractable approximation of the optimization problem in Eq.(1). For arbitrary feasible set, the constrained optimization for $\mathcal{L}_\star(\theta_t - \eta_t u)$ over $u \in U_t$ is generally difficult, especially when U_t is discrete. We therefore apply the majorization–minimization principle [Lange et al., 2000, Mairal, 2013, Lange, 2016] with a standard smoothness assumption [Bottou et al., 2018, Nesterov, 2013].

Assumption 1 (Target smoothness)**.** *The target population loss $\mathcal{L}_\star$ is β-smooth for some $\beta > 0$.*

Under Assumption 1, the target loss admits the following quadratic majorization upper bound:

$$\mathcal{L}_\star(\theta_t - \eta_t u) \le \mathcal{L}_\star(\theta_t) - \eta_t \langle g_\star, u \rangle + \frac{\beta \eta_t^2}{2} \|u\|^2, \tag{2}$$

where $\mathcal{L}_\star(\theta_t)$ is independent of u and the population gradient $g_\star$ is unknown. Hence, minimizing this over $u \in U_t$ and substituting the target batch gradient estimate $\hat{g}_\star$ for the unknown population gradient $g_\star$ yields a Euclidean projection $u_t = \mathrm{Proj}_{U_t}(\hat{g}_\star)$, or equivalently (see derivation in Section B.1):

$$u_t = \arg\min_{u \in U_t} \|u - \hat{g}_\star\|^2, \tag{3}$$

We are now ready to show how different design choices of the feasible set correspond to different methods.

3.3.2 Existing Methods as Special Cases

We now show how several existing training paradigms can be viewed as special cases of our framework. For ease of exposition, we refer to these instantiations as the *Target-Only Update*, the *Full-Training Update*, and the *Global Subset Update*.

Target-Only Update. The least regularized instantiation ignores the training batch when constructing the feasible set and allows the update to move in any direction in parameter space. This corresponds to choosing $U_t^\star := \mathbb{R}^d$. Because there is no directional constraint, Eq.(3) gives $u_t = \hat{g}_\star$. Thus, the Target-Only Update recovers gradient descent on the target batch $B_t^\star$, completely ignoring the training batch B_t. In practice, this corresponds to fine-tuning exclusively on high-quality, task-specific data [Zhou et al., 2023, Gunasekar et al., 2023]. The resulting update is unbiased for $g_\star$, but it can have high variance when the target batch size m is small.

Full-Training Update. At the opposite extreme, the strongest regularization is obtained by allowing only the aggregate training-batch gradient as the update direction. This corresponds to the singleton feasible set $U_t^{\mathrm{tr}} := \{\hat{g}_{\mathrm{tr}}\}$. In this case, Eq.(3) trivially gives $u_t = \hat{g}_{\mathrm{tr}}$. The Full-Training Update recovers standard gradient descent on the general training distribution, as in general-purpose SFT, where the model is trained on a curated mixture without per-step target feedback [Lambert et al., 2024, Grattafiori et al., 2024]. Here, the training batch imposes the strongest possible constraint on the update direction, eliminating any per-step dependence on the target batch.

Global Subset Update. Between these two extremes, existing data selection methods can be expressed by allowing the update to be the average gradient of a selected subset of training samples. For a subset size k, define $U_t^{\mathrm{glob}}(k) := \{g_S = \frac{1}{k}\sum_{i \in S} g_i \colon S \subseteq [n], |S| = k\}$. This feasible set contains all k-sample averages of training gradients from B_t. Under Eq.(3), the resulting update is obtained by choosing $S_t \in \arg\min_{S \subseteq [n], |S|=k} \|g_S - \hat{g}_\star\|^2$, and then setting $u_t = g_{S_t}$. We refer to this instantiation as the *Global* Subset Update, since a single subset S is shared across *all* parameters, in contrast to the *Group-Wise Subset Update* to be introduced next in Section 3.3.3.

The Global Subset Update interpolates between the target-only and Full-Training Updates. Its feasible set $U_t^{\mathrm{glob}}(k)$ is richer than the singleton feasible set $U_t^{\mathrm{tr}} = \{\hat{g}_{\mathrm{tr}}\}$, but remains more restrictive than the unconstrained feasible set $U_t^\star = \mathbb{R}^d$. However, solving for S is a combinatorial optimization

problem over training subsets. Expanding the objective and dropping the term independent of S, the subset optimization is equivalent to minimizing $\|g_S\|^2 - 2\langle g_S, \hat{g}_\star\rangle$. Thus, the objective balances a first-order *alignment* term, $2\langle g_S, \hat{g}_\star\rangle = \frac{2}{k}\sum_{i\in S}\langle g_i, \hat{g}_\star\rangle$, which favors samples aligned with the target gradient, against a second-order *redundancy* penalty, $\|g_S\|^2 = \frac{1}{k^2}\sum_{i,i'\in S}\langle g_i, g_{i'}\rangle$, which discourages selecting highly correlated training gradients. The alignment term decomposes into independent per-sample scores

$$s_i \coloneqq \langle g_i, \hat{g}_\star\rangle,$$

whereas the redundancy penalty couples all samples in S, making exact optimization intractable. Existing data selection methods can therefore be viewed as tractable approximations to this subset optimization problem:

- **Top-k.** Choose the k samples with the largest scores s_i, ignoring the redundancy penalty. This is the cheapest strategy and is effective when samples are not highly correlated [Han and Tsvetkov, 2021, Hu et al., 2024, He et al., 2024].
- **Thresholding.** Keep all samples whose score exceeds a fixed threshold, yielding a variable-size subset [Hu et al., 2025b]. This can be viewed as a variable-cardinality relaxation of the fixed-k feasible set, and it also ignores the redundancy penalty.
- **Greedy construction.** Iteratively construct S_t by adding the sample that most improves the full objective, thereby accounting for redundancy at each step [Wang et al., 2024b]. This better approximates the projection problem, but incurs an additional $O(nk)$ selection cost.

3.3.3 New Data-Regularization Designs via Group-Wise Decomposition

The three instantiations above are only a few points in the broader space of feasible-set designs. In principle, Eq. (3) allows many possible choices of $U_t \subseteq \mathbb{R}^d$ as long as it is independent of target data; our focus here is the ones that actually depend on training data. We introduce a new family of *data regularizers*: feasible sets constructed from the training batch that are more flexible than Global Subset Update, while still restricting target-driven updates to directions supported by training samples.

The motivation is to relax the global coupling imposed by $U_t^{\mathrm{glob}}(k)$, where a single subset S of training samples must be shared across all parameter coordinates. This can be overly restrictive when different parameter groups align with the target signal in different ways. For example, a training sample that provides a useful update for one layer or module may be less informative for another. Thus, rather than selecting one global subset for the entire model, we partition the parameter coordinates into groups and allow each group to choose its own subset of training samples. This gives a controlled relaxation of the feasible set: it weakens the data regularization imposed by the Global Subset Update, while avoiding the fully unconstrained Target-Only Update.

Group-Wise Subset Update. Let $\mathcal{G} = \{G_p\}_{p=1}^P$ be a partition of the parameter indices $[d]$ into P disjoint groups. For any vector $v \in \mathbb{R}^d$, let $v^{(p)}$ denote the subvector indexed by G_p. In particular, $g_i^{(p)}$ and $\hat{g}_\star^{(p)}$ denote the corresponding subvectors of g_i and $\hat{g}_\star$.

For a fixed subset size k, define the group-wise feasible set as

$$U_t^{\mathrm{grp}}(k;\mathcal{G}) \coloneqq \left\{u = (u^{(1)}, \ldots, u^{(P)}) \colon u^{(p)} \in U_t^{(p)}(k) \text{ for all } p \in [P]\right\},$$

where

$$U_t^{(p)}(k) \coloneqq \left\{ g_S^{(p)} \coloneqq \frac{1}{k}\sum_{i\in S} g_i^{(p)} \colon S \subseteq [n], |S| = k \right\}.$$

In other words, each parameter group G_p is allowed to choose its own subset $S_{t,p}$ of training samples. We refer to this instantiation as the *Group-Wise Subset Update*. A practically important special case is the *Layer-Wise Subset Update*, where each group corresponds to the parameters of a single layer.

The Global Subset Update is recovered as the special case in which all groups share the same subset,

$$S_{t,1} = \cdots = S_{t,P}.$$

Therefore,

$$U_t^{\mathrm{glob}}(k) \subseteq U_t^{\mathrm{grp}}(k;\mathcal{G}),$$

with the inclusion typically being strict for nontrivial partitions. Thus, the Group-Wise Subset Update enlarges the feasible set relative to the Global Subset Update. From the data-regularization perspective, this reduces the strength of the regularization imposed by requiring a single global subset, and can reduce approximation bias when that global constraint is too restrictive.

This construction also preserves a useful separability structure. Because the group-wise feasible set factorizes across parameter groups, $U_t^{\mathrm{grp}}(k;\mathcal{G}) = \prod_{p=1}^{P} U_t^{(p)}(k)$, the projection problem in Eq.(3) decomposes into independent per-group problems:

$$\min_{u \in U_t^{\mathrm{grp}}(k;\mathcal{G})} \|u - \hat{g}_\star\|^2 = \sum_{p=1}^{P} \min_{u^{(p)} \in U_t^{(p)}(k)} \|u^{(p)} - \hat{g}_\star^{(p)}\|^2.$$

Equivalently, for each group p, we choose

$$S_{t,p} \in \operatorname*{arg\,min}_{S \subseteq [n], |S|=k} \left\| g_S^{(p)} - \hat{g}_\star^{(p)} \right\|^2,$$

and set

$$u_t^{(p)} = g_{S_{t,p}}^{(p)} = \frac{1}{k} \sum_{i \in S_{t,p}} g_i^{(p)}.$$

Thus, the same approximation strategies used for the Global Subset Update can be applied independently within each parameter group. For example, the group-wise top-k rule scores each training sample by

$$s_i^{(p)} := \langle g_i^{(p)}, \hat{g}_\star^{(p)} \rangle$$

and selects the k samples with the largest scores separately for each group G_p. Thresholding and greedy construction can be adapted analogously.

The partition $\mathcal{G}$, therefore, becomes a new design knob. Coarser partitions impose stronger data regularization by forcing larger portions of the model to share the same subset, while finer partitions allow more flexible target-driven updates. This produces a spectrum of feasible sets between the Global Subset Update and the Target-Only Update, which we analyze through the bias–variance tradeoff in Section 3.4.

3.4 Bias–Variance Tradeoffs

We now formalize the bias–variance tradeoff induced by different choices of the feasible set. From the data-regularization perspective, the feasible set U_t controls the strength of regularization imposed by the training batch. This is analogous to the classical bias–variance tradeoff in statistical estimation [Hastie et al., 2009], but here the relevant complexity is the complexity of the data-induced feasible set rather than the model class.

3.4.1 Data-Regularization Spectrum

The feasible sets introduced above can be organized by their expressiveness. For a fixed subset size k and partition $\mathcal{G}$, we have

$$U_t^{\mathrm{glob}}(k) \subseteq U_t^{\mathrm{grp}}(k;\mathcal{G}) \subseteq U_t^{\star} = \mathbb{R}^d.$$

The first inclusion follows because the Global Subset Update is the special case of the Group-Wise Subset Update in which all parameter groups share the same subset. The second inclusion is immediate because all feasible-set designs are subsets of the unconstrained target-only feasible set. In addition, the Full-Training Update is recovered from the Global Subset Update when $k = n$, since

$$U_t^{\mathrm{glob}}(n) = \left\{ \frac{1}{n} \sum_{i=1}^{n} g_i \right\} = U_t^{\mathrm{tr}}.$$

Thus, by varying the subset size k and the partition $\mathcal{G}$, the proposed framework induces a data-regularization spectrum (Figure 2): moving toward larger feasible sets weakens the regularization imposed by the training batch, reducing approximation bias but increasing statistical variance.

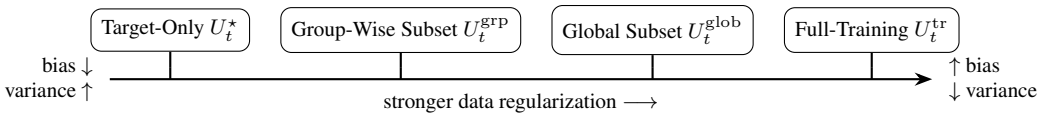

Figure 2: Data regularization spectrum. Different methods correspond to feasible sets of increasing expressiveness (left to right), trading approximation bias for statistical variance.

In particular, Group-Wise Subset Update provides a richer family of data regularizers than the Global Subset Update, allowing a broader design space with different bias–variance tradeoffs.

3.4.2 Formal Characterization of the Tradeoff

We now formalize the bias–variance tradeoff in terms of one-step progress on the target loss. Recall from Eq.(2) that the decrease in target loss is controlled by how well the update direction approximates the target population gradient $g_\star$. We measure this error by the conditional mean squared error (MSE)

$$\mathrm{MSE}(u) \coloneqq \mathbb{E}\left[\|u - g_\star\|^2 \middle| \theta_t\right],$$

where the expectation is over the randomness in the training batch B_t and the target batch $B_t^\star$ sampled at step t. Under this convention, $g_\star = g_\star(\theta_t)$ is fixed, while U_t, $\hat{g}_{\mathrm{tr}}$, $\hat{g}_\star$, and the resulting update u_t are random variables. Formally, we have the following:

Lemma 3.1. *Assume Assumption 1. Fix θ_t and let $0 < \eta_t \le 1/\beta$. For any u with $\mathbb{E}[\|u\|^2 \mid \theta_t] < \infty$,*

$$\mathbb{E}\left[\mathcal{L}_\star(\theta_{t+1}) \mid \theta_t\right] \le \mathcal{L}_\star(\theta_t) - \frac{\eta_t}{2}\|g_\star\|^2 + \frac{\eta_t}{2}\,\mathrm{MSE}(u).$$

We note that Lemma 3.1 holds for any u, and the proof can be found in Section B.2. Now, given a data-induced feasible set U_t, define its approximation bias term as

$$\mathcal{B}(U_t) \coloneqq \mathbb{E}\left[\inf_{u \in U_t} \|u - g_\star\|^2 \middle| \theta_t\right].$$

This quantity measures the best target-gradient approximation achievable within the feasible set, averaged over the randomness in the training batch that constructs U_t. For the projected update u_t obtained from Eq.(3), we define the remaining variance component as

$$\mathcal{V}(u_t) \coloneqq \mathrm{MSE}(u_t) - \mathcal{B}(U_t),$$

so that

$$\mathrm{MSE}(u_t) = \mathcal{B}(U_t) + \mathcal{V}(u_t). \tag{4}$$

The first term captures the "bias" of restricting updates to U_t, while the second term captures the "variance" of choosing an update from U_t using finite-sample target information.

Remark 3.2 (Comparison to Classical Bias and Variance Definitions)**.** *The decomposition in Eq.(4) is adapted to the feasible-set view and is therefore slightly different from the classical bias–variance decomposition of an estimator. Here, $\mathcal{B}(U_t)$ measures the approximation error induced by the feasible set itself, including any randomness in the training-batch-induced feasible set. The variance component $\mathcal{V}(u_t)$ measures the additional error incurred when the update is chosen from U_t using the noisy target-batch estimate $\hat{g}_\star$.*

We are now ready to state our results about the bias and variance of each method. First, we characterize the exact bias–variance decomposition of the Full-Training and Target-Only Updates. For notational convenience, define $\Sigma_{\mathrm{tr}} \coloneqq \mathrm{Cov}(g_i \mid \theta_t)$ and $\Sigma_\star \coloneqq \mathrm{Cov}(g_j^\star \mid \theta_t)$.

Proposition 3.3 (Bias–variance tradeoffs)**.** *Fix θ_t and let $\eta_t = 1/\beta$. Then:*

(i) ***Full-Training Update.*** *For $U_t^{\mathrm{tr}} = \{\hat{g}_{\mathrm{tr}}\}$,*

$$\mathcal{B}(U_t^{\mathrm{tr}}) = \|g_{\mathrm{tr}} - g_\star\|^2 + \frac{\mathrm{tr}(\Sigma_{\mathrm{tr}})}{n}, \qquad \mathcal{V}(u_t^{\mathrm{tr}}) = 0.$$

(ii) ***Target-Only Update.*** *For* $U_t^\star = \mathbb{R}^d$,

$$\mathcal{B}(U_t^\star) = 0, \qquad \mathcal{V}(u_t^\star) = \frac{\mathrm{tr}(\Sigma_\star)}{m}.$$

The proof is deferred to Section B.2. Proposition 3.3 states that the bias of Full-Training Update shrinks as n grows but has an irreducible floor $\|g_{\mathrm{tr}} - g_\star\|^2$ due to the intrinsic distribution-mismatch error between the general training distribution $\mathbb{P}_{\mathrm{tr}}$ and the target distribution $\mathbb{P}_\star$. On the other hand, Target-Only Update achieves zero bias as expected. In contrast, Full-Training Update has zero variance since it does not depend on the target batch, while Target-Only Update has variance $\mathrm{tr}(\Sigma_\star)/m$, which can be large when the target batch size m is small.

Next, we provide explicit variance bounds and bias characterization between Global Subset and Group-Wise Subset Updates. For this, we additionally assume bounded training gradients to control the complexity of the finite feasible sets [Shalev-Shwartz and Ben-David, 2014, Hazan, 2016], and also a standard sub-Gaussian noise assumption [Lan, 2020, Liu et al., 2023]:

Assumption 2 (Bounded gradients)**.** *There exists* $C > 0$ *such that* $\|g_i\| \le C$ *for all* $i \in [n]$.

Assumption 3 (Sub-Gaussian noise)**.** *Noise of target gradient* $\xi := \hat{g}_\star - g_\star$ *is sub-Gaussian with parameter* $\sigma/\sqrt{m}$ *for some* $\sigma > 0$.

Theorem 3.4 (Bias–variance tradeoffs)**.** *Assume Assumptions 2 and 3 and fix* θ_t *and let* $\eta_t = 1/\beta$. *Then for any* $k \in [n]$*:*

(i) ***Global Subset Update.*** *For* $U_t^{\mathrm{glob}}(k)$,

$$\mathcal{V}(u_t^{\mathrm{glob}}(k)) \le \frac{4C\sigma}{\sqrt{m}} \sqrt{2 \log\left(2\binom{n}{k}\right)}.$$

(ii) ***Group-Wise Subset Update.*** *For* $U_t^{\mathrm{grp}}(k;\mathcal{G})$ *where* $\mathcal{G} = \{G_p\}_{p=1}^P$ *contains* P *groups,*

$$\mathcal{V}(u_t^{\mathrm{grp}}(k;\mathcal{G})) \le \frac{4CP\sigma}{\sqrt{m}} \sqrt{2 \log\left(2\binom{n}{k}\right)}.$$

Moreover, Global Subset Update has a higher bias compared to Group-Wise Subset Update:

$$\mathcal{B}(U_t^{\mathrm{grp}}(k;\mathcal{G})) \le \mathcal{B}(U_t^{\mathrm{glob}}(k)).$$

Figure 3 illustrates the decomposition of Theorem 3.4. The proof is again deferred to Section B.2, where we show in the proof that the variance is controlled by $\mathbb{E}[\sup_{u \in U_t^{\mathrm{grp}}(k;\mathcal{G})} |\langle \xi, u \rangle| \mid \theta_t]$, which is in turn upper-bounded by a complexity term that depends on the logarithmic size of the feasible set.

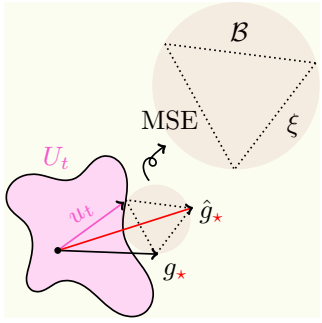

Figure 3: Bias–variance decomposition for a chosen U_t

In terms of bias, we note that $\mathcal{B}(U_t^{\mathrm{grp}}(k;\mathcal{G})) \le \mathcal{B}(U_t^{\mathrm{glob}}(k))$ is simply due to $U_t^{\mathrm{glob}}(k) \subseteq U_t^{\mathrm{grp}}(k;\mathcal{G})$. More generally, for two groups $\mathcal{G}, \mathcal{G}'$ where $\mathcal{G}'$ is a finer partition of $\mathcal{G}$, we have $U_t^{\mathrm{grp}}(k;\mathcal{G}) \subseteq U_t^{\mathrm{grp}}(k;\mathcal{G}')$, and hence $\mathcal{B}(U_t^{\mathrm{grp}}(k;\mathcal{G}')) \le \mathcal{B}(U_t^{\mathrm{grp}}(k;\mathcal{G}))$. On the other hand, for any two $k \ne k'$, it is almost always the case that no inclusion relation exists between $U_t^{\mathrm{glob}}(k)$ and $U_t^{\mathrm{glob}}(k')$ (or $U_t^{\mathrm{grp}}(k;\mathcal{G})$ and $U_t^{\mathrm{grp}}(k';\mathcal{G})$), and hence it is generally difficult to compare the bias for different k. However, the same intuition still holds: the larger the feasible is, the lower the bias might be. For instance, when $k = n$, we have $U_t^{\mathrm{tr}} = U_t^{\mathrm{glob}}(n) = U_t^{\mathrm{grp}}(n;\mathcal{G})$, degenerating to the singleton set; when $k = n/2$, the feasible set $U_t^{\mathrm{glob}}(k)$ has $\binom{n}{n/2}$ candidates for approximating $g_\star$, boosting the chance of approximating $g_\star$ better and hence achieving a lower bias.

Summary. Among all the methods, Full-Training Update imposes the strongest data regularization: it is stable, but may incur substantial bias when the training and target distributions are misaligned. Target-Only Update imposes no data regularization: it is unbiased for the target gradient, but can be

unstable when m is small. Global Subset Update interpolates these two, offering a new bias–variance tradeoff, and group-wise decomposition further reduces approximation bias by relaxing the global subset constraint, offering a spectrum over bias–variance tradeoffs with different choices of group partition.

Takeaway: *Larger feasible sets reduce bias but increase variance*, since the update has more freedom to align both with $g_\star$ and **also with noise** ξ. Full-Training is stable but biased; Target-Only is unbiased but unstable when m is small; Global and Group-Wise Subset Updates interpolate between them, with the partition $\mathcal{G}$ and subset size k providing additional design knobs.

3.5 Generalization Beyond Two-Distribution Setting

Although we developed the framework in the two-distribution setting above, the underlying principle is more general. At its core, the data regularization principle rests on the following structure:

(i) *Training signal* that can be optimized reliably but might introduce bias, and

(ii) *Target signal* that is too noisy or costly to optimize directly.

In the two-distribution setting, the target batch provides the noisy signal for optimizing the target objective, while the training batch induces the feasible set that stabilizes the update.

Other common post-training paradigms also admit this structure: for instance, in reinforcement learning, the target signal of interest is the expected reward, but directly optimizing it via policy gradient is high-variance. Training signals such as PPO's clipped loss [Schulman et al., 2017] or GRPO's group-relative estimator [Shao et al., 2024], on the other hand, stabilize training at the cost of optimization bias. We demonstrate and apply these principles in the experiments (Sections 5.2 and 5.3).

4 Efficient Realization of Dr. Post-Training Under Memory Constraints

Modern LLM training infrastructures are heavily constrained by memory bottlenecks. Efficiently realizing the proposed method under such tight memory limits, therefore, requires nontrivial system-level optimizations. In this section, we discuss the optimizations that make Dr. Post-Training practical and compatible with modern infrastructures.

Specifically, we will start by revisiting the memory footprint of standard training (Section 4.1) that grounds the later discussions in three aspects. First, in Section 4.2, we address the computational overhead of data-regularized updates with a carefully designed tensor lifetime schedule that can efficiently reuse the intermediate variables in standard forward and backward passes with negligible additional memory cost. Second, in Section 4.3, we investigate the computational costs of the key per-sample scoring step incurred by the data-regularized updates, develop several implementations with different compute–memory trade-offs, and identify the most efficient choice for different regimes. Third, in Section 4.4, we show that the resulting design remains compatible with modern memory-saving training techniques such as LoRA, MeSO, activation checkpointing, and gradient accumulation.

Overall, this section shows that Dr. Post-Training is not only algorithmically appealing, but can also be implemented efficiently under the memory constraints that dominate modern post-training.

4.1 Preliminary: Memory Footprint in Standard Training

Here, we provide a quick overview of the memory footprint in a standard training step that consists of a forward and a backward pass, where we simplify the transformer model into a stack of L MLP linear layers for discussion convenience.[2] Let the model with d total parameters $\theta_t = (\theta_t^{(1)}, \ldots, \theta_t^{(L)}) \in \mathbb{R}^d$, with each layer $\theta_t^{(l)}$ having d/L parameters and activations of dimension $\sqrt{d/L}$ for simplicity. We use

[2]Throughout this section, "layer" refers to an individual linear module (e.g., `nn.Linear` in PyTorch), not a full transformer block. A single transformer block contains multiple linear modules (query, key, value, output projections, and MLP layers). Data regularization targets these linear modules, which account for the vast majority of transformer parameters; the treatment of embedding layers and other non-linear modules is discussed in Section C.6.

both the vector form $\theta_t^{(l)} \in \mathbb{R}^{d/L}$ and the matrix form $W_t^{(l)} \in \mathbb{R}^{\sqrt{d/L}\times\sqrt{d/L}}$ interchangeably, related by $\theta_t^{(l)} = \mathrm{vec}(W_t^{(l)})$; the same applies to weight gradients in matrix form $G^{(l)} \in \mathbb{R}^{\sqrt{d/L}\times\sqrt{d/L}}$ with $g^{(l)} = \mathrm{vec}(G^{(l)})$.

4.1.1 Forward and Backward Pass

Given a training batch $B_t = \{z_i\}_{i=1}^n$ of sequences of length T, a standard training step first performs a forward pass to evaluate the batch loss $\ell := \sum_{i=1}^n \ell_i$ with $\ell_i := \ell(\theta_t; z_i)$, caching intermediate activations, then a backward pass to compute the batch gradient $\hat{g}_{\mathrm{tr}} = (\hat{g}_{\mathrm{tr}}^{(1)}, \ldots, \hat{g}_{\mathrm{tr}}^{(L)})$. Figure 4 illustrates both passes at a single layer l along with the resulting memory footprint. For simplicity, we omit the dimension of sequence length in the illustration.

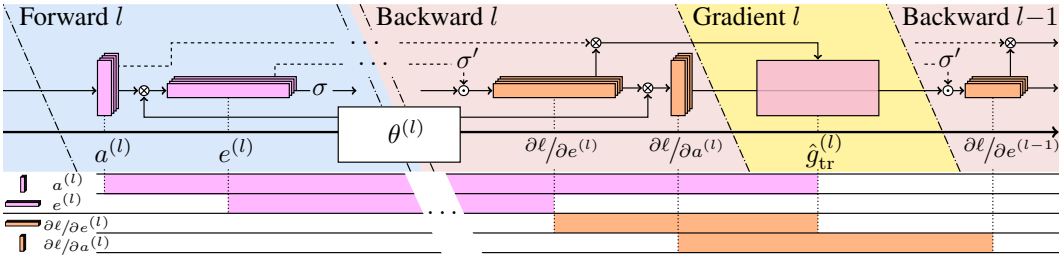

Figure 4: **Top.** Forward pass and backward pass (activation and weight gradient) computation graph: both $\longrightarrow$ and $\dashrightarrow$ denote *computational dependency*; $\dashrightarrow$ indicates that the dependency runs across the boundary between forward and backward. **Bottom.** Memory footprint: a tensor can be released once no remaining operation depends on it.

Forward pass. Each layer l computes the output (pre-activation) of the layer $e_{i,\tau}^{(l)} = W_t^{(l)} a_{i,\tau}^{(l)}$ from the input activation $a_{i,\tau}^{(l)} \in \mathbb{R}^{\sqrt{d/L}}$ for each sample $i \in [n]$ and token $\tau \in [T]$, then applies an activation function σ: $a_{i,\tau}^{(l+1)} = \sigma(e_{i,\tau}^{(l)})$. In practice, all nT vectors are batched as columns of an activation matrix, so each layer reduces to a single matrix multiplication. All activations and pre-activations $\{a_{i,\tau}^{(l)}, e_{i,\tau}^{(l)}\}_{i,\tau}$ are *cached* for later use.

Backward pass. The activation gradient propagates backward layer by layer. At layer l, the pre-activation gradient $\partial\ell/\partial e_{i,\tau}^{(l)}$ arrives for each sample i and token τ. The activation gradient propagates to layer $l-1$ via $\partial\ell/\partial e_{i,\tau}^{(l-1)} = \sigma'(e_{i,\tau}^{(l-1)}) \odot \partial\ell/\partial a_{i,\tau}^{(l)}$ where $\partial\ell/\partial a_{i,\tau}^{(l)} = W_t^{(l)\top} \partial\ell/\partial e_{i,\tau}^{(l)}$ and $\odot$ denotes the element-wise product. The batch weight gradient is then computed as $\hat{g}_{\mathrm{tr}}^{(l)} = \frac{1}{n}\sum_{i=1}^n \sum_{\tau=1}^T (\partial\ell/\partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)} \in \mathbb{R}^{d/L}$, where $\otimes$ denotes the Kronecker product. For later reference, we write the per-sample weight gradient as $g_i^{(l)} := \sum_{\tau=1}^T (\partial\ell/\partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$, so that $\hat{g}_{\mathrm{tr}}^{(l)} = \frac{1}{n}\sum_{i=1}^n g_i^{(l)}$. In practice, however, standard backpropagation does not explicitly form each $g_i^{(l)}$. Instead, it stacks all token-level activations and pre-activation gradients across the batch and evaluates the above double sum with a single batched matrix multiplication, which directly returns the aggregated batch gradient $\hat{g}_{\mathrm{tr}}^{(l)}$. Thus, standard training computes the *batch gradient* without materializing separate *per-sample weight gradients* in memory.

4.1.2 Memory Footprint of Key Tensors during Training

During each step of the standard training update, a tensor can be released as soon as no remaining computation depends on it. As the goal is to compute $\hat{g}_{\mathrm{tr}}$, a key observation is that during backward pass at layer $l-1$, both $a^{(l)}$ and $\partial\ell/\partial e^{(l)}$ become free once the batch gradient $\hat{g}_{\mathrm{tr}}^{(l)}$ is assembled (Figure 4). Moreover, $\partial\ell/\partial e^{(l+1)}$ is released before $\partial\ell/\partial e^{(l)}$ is allocated, so its buffer is reused and the memory footprint of pre-activation gradients remains constant across layers.

Algorithm 4.1 presents the pseudocode of one standard training step with memory management, which recovers the Full-Training Update. For simplicity, we hide the computation of $a_i^{(l)}$, $e_i^{(l)}$, their gradient, and also their dependency on each other in the backward function call. The creation and release of key tensors of interest are annotated with memory cost ($+$ allocation, $-$ release) to be compared with later algorithms. We note that when the training data is $B_t^\star$ instead of B_t, we recover Target-Only Update.

Algorithm 4.1: Standard Training Update

Data: Model θ_t, training data $\{z_i\}_{i=1}^n$, learning rate η_t
Result: Updated model parameters θ_{t+1}

$(\ell, \{a_i^{(l)}, e_i^{(l)}\}_{l,i}) \leftarrow$ `Forward`$(\theta_t, \{z_i\}_{i=1}^n)$ // $+O(2nT\sqrt{dL})$
$u_t \leftarrow 0 \in \mathbb{R}^d$ // $+O(d)$
for $l = L, \ldots, 1$ **do**
  `/* Allocate` $\{\partial\ell/\partial e_i^{(l)}\}_{i=1}^n$`, release` $\{e_i^{(l)}\}_{i=1}^n$`; memory remains constant */`
  $\{\partial\ell/\partial e_i^{(l)}\}_{i=1}^n \leftarrow$ `Backward`$(\ell, \theta_t^{(l)})$ // $+O(nT\sqrt{d/L}) - O(nT\sqrt{d/L})$
  $u_t^{(l)} \leftarrow \frac{1}{n}\sum_{i=1}^n \sum_{\tau=1}^T (\partial\ell/\partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$
  `Release`$(a_i^{(l)}, \partial\ell/\partial e_i^{(l)})$ for all $i \in [n]$ // $-O(2nT\sqrt{d/L})$
$\theta_{t+1} \leftarrow \theta_t - \eta_t u_t$
`Release`(u_t) // $-O(d)$
return θ_{t+1}

4.2 Customized Tensor Lifetime Scheduling for Efficient Data-Regularized Update

We now turn to a main system challenge introduced by the additional computational dependencies in data-regularized updates. Recall that, unlike standard training, our method must first determine the final update direction by solving a subset-selection problem based on per-sample scores, and only then assemble the corresponding gradient update. This creates additional computational dependencies on intermediate per-sample quantities that are not preserved in standard backpropagation.

Naive implementations. A naive implementation is a *two-pass* approach: use one forward-backward pass to compute the per-sample scores and determine the selected subset, and then run a second forward-backward pass to compute the final gradient update using only the selected subset[3]. This approach is simple and memory-efficient because each pass follows the standard training schedule, but it substantially increases runtime by effectively duplicating the backward computation.

Another naive implementation is a *retain-graph* approach: retain the computation graph (e.g., by calling `retain_graph=True` in `PyTorch`) during backpropagation and keep all intermediate tensors alive so that the same computation graph can be reused for both scoring and gradient assembly. While this avoids the second pass, it is impractical at LLM scale, since retaining the full graph together with the per-sample quantities needed by our method leads to prohibitive memory overhead.

4.2.1 Customized Tensor Lifetime Scheduling

We develop a customized tensor lifetime scheduling to obtain the data-regularized gradient update within *one pass* while avoiding significant memory overhead.

For convenience, we consider *layer-aligned* partition in the rest of the discussion, where each group contains one or more layers. We direct interested readers to Section C.1 for a discussion on general partitions, where each group might contain partial parameters of a layer.

A starting trick: merging training and target batch. We start by addressing a problem that the training gradients and target gradients are typically calculated in separate passes, which prevents effective tensor lifetime scheduling in one pass. We address this problem by realizing that these

[3]For example, an existing online data selection method [Wang et al., 2024b] adopts a two-pass implementation. See their implementation in https://github.com/Jiachen-T-Wang/GREATS/ and an in-depth discussion in Section C.2.

gradients can be obtained from a *merged batch* with both training and target samples. Specifically, suppose a training step uses a training batch $B_t = \{z_i\}_{i=1}^n$ and a target batch $B_t^\star = \{z_j^\star\}_{j=1}^m$. Rather than processing them in separate passes, we concatenate them into one batch of size $N = n + m$ and define the merged loss

$$\ell := \sum_{i=1}^{n} \ell(\theta_t; z_i) + \sum_{j=1}^{m} \ell(\theta_t; z_j^\star).$$

Because different samples do not share activations, a single forward-backward pass on this merged loss produces all *per-sample* backward signals for both the training samples and the target samples. From these, we can extract the quantities needed for constructing the data-regularized update direction.

Remark 4.1. *Doing forward and backward pass on the merged-batch loss $\ell := \sum_{i=1}^n \ell_i + \sum_{j=1}^m \ell_j^\star$ is an existing trick in the literature [Pandya et al., 2025, Wang et al., 2024b]. We note that this extends to non-loss-gradient target signals for an arbitrary θ_t-differentiable $f(\{z_j^\star\}_{j=1}^m; \theta_t)$ (Section 3.2): backward pass on the merged batch with $\ell := \sum_{i=1}^n \ell_i + f(\{z_j^\star\}_{j=1}^m; \theta_t)$ yields both the per-sample training gradients and the target gradient $\nabla_\theta f$ in a single pass; however, when f depends on the training samples z_i, the gradient information is mixed and separate passes are required.*

The computational dependencies in data-regularized update. Next, given the merged batch where we can obtain the per-sample quantities from one pass, we carefully examine the computational dependencies for obtaining the data-regularized updates. For both Global Subset Update and Group-Wise Subset Update, the algorithm can be decomposed into three parts:

1. **Scoring:** compute the score of each training sample against the target signal (at group level);
2. **Subset selection:** determine the selected subset S_t (or $S_{t,p}$ for each group G_p);
3. **Update assembly:** assemble the final update direction by averaging the gradients over the selected subset.

The key issue is that both *scoring* and *update assembly* depend on per-sample quantities, while standard training only materializes the aggregated batch gradient $\hat{g}_{\mathrm{tr}}$. In particular, the per-sample gradient $g_i^{(l)} = \sum_{\tau=1}^T (\partial \ell / \partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$ is never explicitly stored in standard backpropagation. Yet for our method, it enters twice: first in the score computation, and later in the assembly of the selected subset. This is exactly the source of the extra dependencies that need to be handled.

A crucial observation is that both scoring and update assembly ultimately depend on the same local information at each layer, namely the pair $(a^{(l)}, \partial \ell / \partial e^{(l)})$. Once this pair is available, one can either materialize the corresponding per-sample gradients $g_i^{(l)}$ explicitly, or compute the required scores and selected averages directly from it. This suggests that the fundamental scheduling question is not whether to preserve the entire computation graph, but rather how to retain this minimal local information long enough for all downstream computations to be completed.

Tensor lifetime scheduling. Given the computational dependencies, we now introduce the proposed tensor lifetime scheduling.

The key idea is to *swap* the forward-cached tensor $e^{(l)}$ with the backward-generated tensor $\partial \ell / \partial e^{(l)}$ at each layer. Recall from Section 4.1 that in standard training the forward pass caches $(a^{(l)}, e^{(l)})$, and during backward, $e^{(l)}$ is consumed to produce $\partial \ell / \partial e^{(l)}$. After this point, $e^{(l)}$ is no longer needed. Therefore, instead of releasing everything immediately after the layer gradient is computed, we retain the pair $(a^{(l)}, \partial \ell / \partial e^{(l)})$ for later scoring and update assembly, while discarding $e^{(l)}$. Since $\partial \ell / \partial e^{(l)}$ has the same shape as $e^{(l)}$, this replacement preserves the per-layer memory footprint up to lower-order bookkeeping costs.

Applying this swap iteratively across layers yields a one-pass schedule with the following structure:

1. run one forward pass on the merged batch and cache $(a^{(l)}, e^{(l)})$ as in standard training;
2. during backward, replace each cached $e^{(l)}$ with $\partial \ell / \partial e^{(l)}$, so that after the backward pass the retained tensors are $(a^{(l)}, \partial \ell / \partial e^{(l)})$ for all layers;
3. use these retained pairs to compute scores, determine the selected subset(s), and assemble the final update direction;

4. release the retained tensors once all groups depending on them have been resolved.

This schedule differs fundamentally from `retain_graph=True`: we do *not* keep the full autograd graph alive, but only the minimal local tensors needed for later computation. At the same time, unlike the two-pass implementation, we do *not* rerun the backward pass. The result is a one-pass implementation that stays close to the peak memory cost of standard training while avoiding a near-doubling of runtime.

When can a group be resolved? The remaining question is when a group can be finalized and its tensors released. This depends on the partition granularity. For a general partition $\mathcal{G} = \{G_p\}_{p=1}^{P}$, the subset $S_{t,p}$ for group G_p cannot be determined until all score contributions from the layers intersecting G_p have been accumulated. Only after that can the corresponding update $u_t^{(p)}$ be assembled. Therefore, a group can be released only after all of its constituent layers have finished both scoring and update assembly.

This reveals a direct connection between the statistical design and the systems design: finer partitions not only give more flexible update rules, but also allow earlier release of tensors and hence a memory footprint closer to standard training[4].

4.2.2 Case Study: Group Granularity

We illustrate the above scheduling with two extreme cases.

Global Subset Update. For Global Subset Update, all parameters share a single global subset S_t. Consequently, the score of each training sample must aggregate contributions from *all* layers before S_t can be determined. This means the retained pairs $(a^{(l)}, \partial\ell/\partial e^{(l)})$ must remain alive across the entire model until scoring is complete. After the global subset S_t is determined, the final update is assembled by revisiting each retained layer and averaging the per-sample gradients of the selected samples (See Algorithm 4.2). Compared with standard training, the peak memory remains close, but the high-memory period now spans almost the entire backward-and-selection phase (Middle row in Figure 5).

Layer-Wise Subset Update. At the other extreme, for Layer-Wise Subset Update, each layer forms its own group. In this case, the score computation, subset selection, and update assembly for layer l depend only on the local tensors of that same layer. Therefore, once the backward pass reaches layer l, we can immediately compute the layer-wise scores, determine $S_{t,l}$, assemble the update $u_t^{(l)}$, and release all retained tensors for that layer before moving to layer $l-1$ (See Algorithm 4.3). As a result, the tensor lifetime schedule almost exactly matches that of standard training, and the memory footprint is closest to the standard baseline (Bottom row in Figure 5).

Intermediate granularities. More generally, if each group contains a small number of consecutive layers, then a group's tensors only need to be retained until the backward pass finishes those layers. This gives a smooth interpolation between the two extremes above. In particular, smaller groups allow earlier release and lower sustained memory usage, whereas larger groups delay release and make the schedule resemble Global Subset Update.

Remark 4.2. *One benefit of more aggressive memory release (as in Layer-Wise Subset Update) is that, in practice, when layer sizes are not uniform, later in the backward pass, encountering a larger layer provides enough free memory to process it.*

In summary, the proposed customized tensor lifetime schedule resolves a core system challenge of data-regularized updates: it avoids the runtime overhead of a two-pass implementation and the prohibitive memory cost of retaining the full graph, while enabling a practical one-pass realization under the memory constraints of modern LLM training.

[4]It is worth noting that the peak memory cost remains similar regardless of the partition granularity, since the peak memory is achieved right after the forward pass. However, finer granularity becomes helpful when considering other memory-saving techniques, such as the activation checkpointing discussed in Section 4.4.

Algorithm 4.2: Global Subset Update

Data: Model θ_t, training data $\{z_i\}_{i=1}^n$, target data $\{z_j^\star\}_{j=1}^m$, learning rate η_t
Result: Updated model parameters θ_{t+1}

$(\ell, \{a_i^{(l)}, e_i^{(l)}\}_{l,i} \cup \{a_j^{\star(l)}, e_j^{\star(l)}\}_{l,j}) \leftarrow$ `Forward`$(\theta_t, \{z_i\}_{i=1}^n \cup \{z_j^\star\}_{j=1}^m)$ // $+O(2NT\sqrt{dL})$
for $l = L, \ldots, 1$ **do** // `Retain all pairs via swapping`
  $\{\partial\ell/\partial e_i^{(l)}\}_{i=1}^n \cup \{\partial\ell/\partial e_j^{\star(l)}\}_{j=1}^m \leftarrow$ `Backward`$(\ell, \theta_t^{(l)})$
`/* Post-hoc: scoring from retained activations */`
for $l = L, \ldots, 1$ **do** // `Peak:` $2NT\sqrt{dL}$
  $g_i^{(l)} \leftarrow \sum_{\tau=1}^T (\partial\ell/\partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$ for each $i \in [n]$ // $+O(n\sqrt{d/L})$
  $\hat{g}_\star^{(l)} \leftarrow \frac{1}{m}\sum_{j=1}^m \sum_{\tau=1}^T (\partial\ell/\partial e_{j,\tau}^{\star(l)}) \otimes a_{j,\tau}^{\star(l)}$ // $+O(\sqrt{d/L})$
  `Release`$(a_j^{\star(l)}, \partial\ell/\partial e_j^{\star(l)})$ for all $j \in [m]$ // $-O(2mT\sqrt{d/L})$
  $s_i^{(l)} \leftarrow \langle \hat{g}_\star^{(l)}, g_i^{(l)} \rangle$ for each $i \in [n]$
  `Release`$(g_i^{(l)}, \hat{g}_\star^{(l)})$ for all $i \in [n]$ // $-O((n+1)\sqrt{d/L})$
$s_i \leftarrow \sum_{l=1}^L s_i^{(l)}$ for all $i \in [n]$ // `Global scores`
$S_t \leftarrow$ `Select-S`$(\{s_i\}_{i=1}^n, k)$
`/* Post-hoc: assemble gradients from retained activations */`
$u_t \leftarrow 0 \in \mathbb{R}^d$ // $+O(d)$
for $l = L, \ldots, 1$ **do**
  $u_t^{(l)} \leftarrow \frac{1}{k}\sum_{i \in S_t} \sum_{\tau=1}^T (\partial\ell/\partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$
  `Release`$(a_i^{(l)}, \partial\ell/\partial e_i^{(l)})$ for all $i \in [n]$ // $-O(2nT\sqrt{d/L})$
$\theta_t \leftarrow \theta_t - \eta_t u_t$
`Release`(u_t) // $-O(d)$
return θ_{t+1}

Algorithm 4.3: Layer-Wise Subset Update

Data: Model θ_t, training data $\{z_i\}_{i=1}^n$, target data $\{z_j^\star\}_{j=1}^m$, learning rate η_t
Result: Updated model parameters θ_{t+1}

$(\ell, \{a_i^{(l)}, e_i^{(l)}\}_{l,i} \cup \{a_j^{\star(l)}, e_j^{\star(l)}\}_{l,j}) \leftarrow$ `Forward`$(\theta_t, \{z_i\}_{i=1}^n \cup \{z_j^\star\}_{j=1}^m)$ // $+O(2NT\sqrt{dL})$
$u_t \leftarrow 0 \in \mathbb{R}^d$ // $+O(d)$
for $l = L, \ldots, 1$ **do**
  $\{\partial\ell/\partial e_i^{(l)}\}_{i=1}^n \cup \{\partial\ell/\partial e_j^{\star(l)}\}_{j=1}^m \leftarrow$ `Backward`$(\ell, \theta_t^{(l)})$
  $g_i^{(l)} \leftarrow \sum_{\tau=1}^T (\partial\ell/\partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$ for each $i \in [n]$ // $+O(n\sqrt{d/L})$
  $\hat{g}_\star^{(l)} \leftarrow \frac{1}{m}\sum_{j=1}^m \sum_{\tau=1}^T (\partial\ell/\partial e_{j,\tau}^{\star(l)}) \otimes a_{j,\tau}^{\star(l)}$ // $+O(\sqrt{d/L})$
  $s_i^{(l)} \leftarrow \langle \hat{g}_\star^{(l)}, g_i^{(l)} \rangle$ for each $i \in [n]$
  $S_{t,l} \leftarrow$ `Select-S`$(\{s_i^{(l)}\}_{i=1}^n, k)$
  $u_t^{(l)} \leftarrow \frac{1}{k}\sum_{i \in S_{t,l}} g_i^{(l)}$
  `Release`$(g_i^{(l)}, \hat{g}_\star^{(l)})$ for all $i \in [n]$ // $-O((n+1)\sqrt{d/L})$
  `Release`$(a_i^{(l)}, \partial\ell/\partial e_i^{(l)}, a_j^{\star(l)}, \partial\ell/\partial e_j^{\star(l)})$ for all $i \in [n], j \in [m]$ // $-O(2NT\sqrt{d/L})$
$\theta_t \leftarrow \theta_t - \eta_t u_t$
`Release`(u_t) // $-O(d)$
return θ_{t+1}

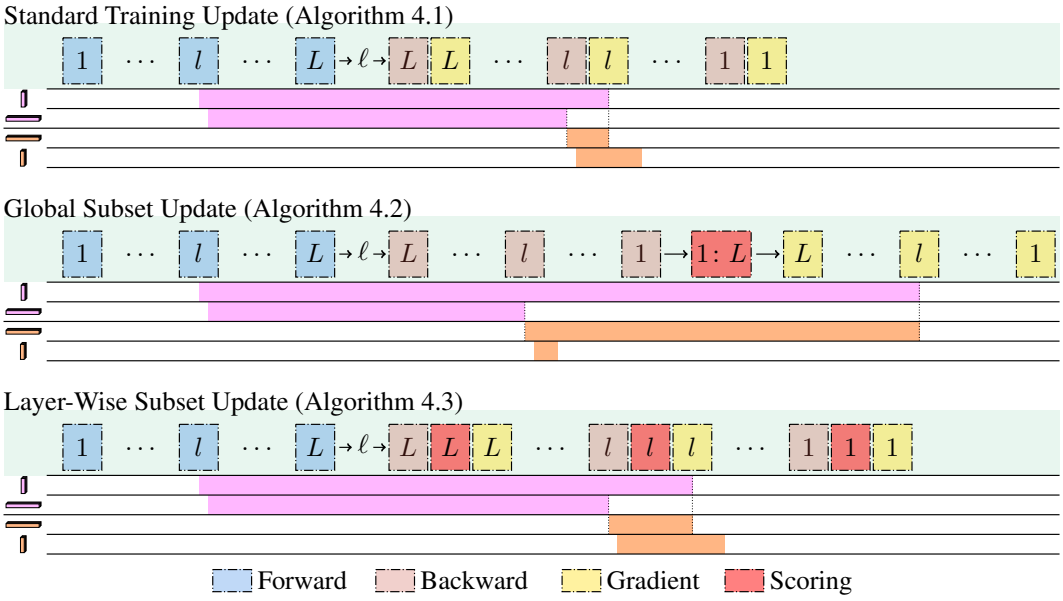

Figure 5: Memory footprint comparison. **Top:** Standard Training Update (Algorithm 4.1). **Middle:** Global Subset Update (Algorithm 4.2). **Bottom:** Layer-Wise Subset Update (Algorithm 4.3). All three methods achieve comparable peak memory per layer, but Global Subset Update must retain both $a^{(l)}$ and $\partial \ell / \partial e^{(l)}$ for all l throughout the backward pass until the global subset is determined, while standard training and Layer-Wise Subset Update release them on-the-fly.

4.3 Efficient Implementations of Per-Sample Scoring

We next study efficient implementations of per-sample scoring, which constitutes a major computational overhead introduced by data-regularized updates. While the tensor lifetime schedule in Section 4.2 resolves the memory-management challenge of preserving the necessary intermediate quantities within one pass, the overall efficiency of the method still depends critically on how the per-sample scores are computed.

In this subsection, we revisit two existing methods for score computation and introduce two new variants. We then provide a systematic complexity analysis in terms of the key training hyperparameters. This analysis reveals that the relative advantages of different implementations depend strongly on the hyperparameter regime, and that this regime-dependent comparison has not been fully characterized in the existing literature. Among these methods, one of our proposed variants—an approximate scoring method—achieves the best computational efficiency across all regimes.

Throughout this section, we focus on the score restricted to one particular layer, $s_i^{(l)}$, since when a parameter group spans multiple layers, the total score is simply the sum of the scores from those layers.

Table 1: Computational overhead of scoring per layer for n training samples and m target samples, where $N = n + m$, T denotes the sequence length, and κ is the compressed gradient dimension ($\kappa \ll d/L$). All costs listed below represent additional overhead beyond standard backpropagation. We assume fully batched execution in calculating the complexity; micro-batching over samples or tokens can trade parallelism for lower memory, which further favors the proposed PIP method.

Scoring method	**FLOPs per layer**	**Memory per layer (entries)**
Direct	$2NT(d/L) + n(d/L)$	$(n+1)(d/L)$
GIP	$4nmT^2\sqrt{d/L}$	$2nmT^2$
PIP	$2NT(d/L) + nT\sqrt{d/L}$	$d/L + nT\sqrt{d/L}$
Compressed	$O(NT\kappa) + (2n+m)\kappa$	$(n+1)\kappa$

4.3.1 Existing Methods: Direct and Ghost Inner Product (GIP)

We first revisit two existing methods, *Direct* and *Ghost Inner Product* (GIP).

Direct. Direct is a naive approach for the score computation. It directly materializes each per-sample gradient $g_i^{(l)} = \sum_\tau (\partial \ell / \partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$ and the target gradient $\hat{g}_\star^{(l)}$ first, then computes the score as $s_i^{(l)} = \langle \hat{g}_\star^{(l)}, g_i^{(l)} \rangle$. Existing literature criticizes this naive approach for materializing $O(n)$ per-sample gradient vectors and incurring high computational costs, which led to the GIP introduced below [Wang et al., 2024b]. However, we demonstrate that the comparison between Direct and GIP is more nuanced than the literature suggested when considering the sequence length T and the number of target samples m.

Ghost Inner Product (GIP) [Wang et al., 2024b]. GIP avoids the full gradient materialization (thus the name "Ghost") by evaluating the same inner product in an alternative contraction order from $a^{(l)}$ and $\partial \ell / \partial e^{(l)}$: contracting over the model dimension instead of the token dimension yields $T \times T$ cross-correlations. Treating $\partial \ell / \partial e^{(l)}, a^{(l)} \in \mathbb{R}^{\sqrt{d/L} \times T}$ as matrices whose columns are the per-token vectors, the score is

$$s_i^{(l)} = \frac{1}{m} \sum_{j=1}^{m} \left\langle \left(\frac{\partial \ell}{\partial e_i^{(l)}} \right)^\top \frac{\partial \ell}{\partial e_j^{\star(l)}}, (a_i^{(l)})^\top a_j^{\star(l)} \right\rangle,$$

where each factor is a $T \times T$ matrix product computed over all nm training-target pairs. While this avoids gradient materialization, it introduces quadratic dependence on T and linear dependence on m.

Comparison between Direct and GIP. As shown in Table 1, GIP is competitive only when m and T are relatively small. By comparing the leading terms in the computational complexity, GIP is preferable to Direct only when $mT \lesssim \sqrt{d/L}/2$. A similar trend holds for the memory cost comparison as well.

4.3.2 Per-Token Inner Product (PIP)

The comparison shows that GIP suffers from long context (large T) while Direct suffers from large parameter size per layer (large d/L). We now introduce a third method, *Per-Token Inner Product* (PIP), that interpolates between these two methods and provides a more favorable computation cost trade-off in an important regime.

Observing that the GIP differs from Direct by swapping the order of some linear operators (sum and inner product) in calculating the score $s_i^{(l)}$, our proposed PIP applies a different order. Let $\hat{G}_\star^{(l)} \in \mathbb{R}^{\sqrt{d/L} \times \sqrt{d/L}}$ denote the matrix form of the *aggregated* target gradient $\hat{g}_\star^{(l)}$. Then the score can be written as

$$s_i^{(l)} = \sum_{\tau=1}^{T} \left(\frac{\partial \ell}{\partial e_{i,\tau}^{(l)}} \right)^\top \hat{G}_\star^{(l)} a_{i,\tau}^{(l)}.$$

This expression factors the computation into a matrix-vector product $\hat{G}_\star^{(l)} a_{i,\tau}^{(l)} \in \mathbb{R}^{\sqrt{d/L}}$, followed by a dot product with $\partial \ell / \partial e_{i,\tau}^{(l)}$. Importantly, this avoids materializing the full per-sample training gradient $g_i^{(l)}$: the score is accumulated directly from token-level quantities that are already available under the tensor lifetime schedule.

Comparison with existing methods. As shown in Table 1, the complexities of PIP remain linear in the sequence length T, and therefore avoid the quadratic dependence on T that makes GIP unattractive for long-context training. Compared with Direct, PIP has the same dominant FLOPs term $2NT(d/L)$, but replaces the lower-order term $n(d/L)$ with $nT\sqrt{d/L}$. Therefore, when $T \lesssim \sqrt{d/L}$, PIP is more efficient than Direct in both FLOPs and memory. Intuitively, in this regime, it is cheaper to work directly with token-level quantities than to first materialize n full per-sample gradients. When T becomes large, however, this advantage disappears: the additional $nT\sqrt{d/L}$ term dominates, and

Direct becomes preferable. Thus, PIP occupies an intermediate regime between GIP and Direct: it improves substantially over GIP for long sequences, and improves over Direct when the context length is moderate relative to the per-layer hidden dimension.

4.3.3 Compressed Scoring

All three methods above compute the score exactly, and therefore their cost necessarily depends on the gradient dimension d/L. We now introduce a fourth method, *Compressed Scoring*, which reduces this cost by approximately computing the score in a low-dimensional compressed space.

Let $\Pi^{(l)} : \mathbb{R}^{d/L} \to \mathbb{R}^{\kappa}$ be a compression map with $\kappa \ll d/L$. Instead of directly computing the full per-sample training gradient $g_i^{(l)}$ and target gradient $\hat{g}_\star^{(l)}$, we first compute their compressed versions

$$\tilde{g}_i^{(l)} \coloneqq \Pi^{(l)}(g_i^{(l)}) \in \mathbb{R}^{\kappa}, \qquad \tilde{g}_\star^{(l)} \coloneqq \Pi^{(l)}(\hat{g}_\star^{(l)}) \in \mathbb{R}^{\kappa},$$

and then approximate the score by

$$s_i^{(l)} \approx \langle \tilde{g}_\star^{(l)}, \tilde{g}_i^{(l)} \rangle.$$

The key point is that $\tilde{g}_i^{(l)}$ and $\tilde{g}_\star^{(l)}$ can be computed directly from the retained token-level quantities $(a^{(l)}, \partial \ell / \partial e^{(l)})$, without ever materializing the full d/L-dimensional gradients. In particular, since $g_i^{(l)} = \sum_{\tau=1}^{T} (\partial \ell / \partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)}$, state-of-the-art gradient compression methods in data attribution literature [Hu et al., 2025a, Choe et al., 2025] can exploit this outer-product structure and apply the projection $\Pi^{(l)}$ in a factorized manner (see Section C.3 for more details). As a result, the compressed per-sample gradient can be obtained from the same retained local tensors used throughout Section 4.2, but with cost scaling only in the compressed dimension κ.

Comparison with exact methods. As shown in Table 1, Compressed Scoring has per-layer cost $O(NT\kappa) + (2n + m)\kappa$ and memory cost $(n + 1)\kappa$, which removes the dependence on the original gradient dimension d/L from both FLOPs and memory. Since $\kappa \ll d/L$, this makes Compressed Scoring strictly more efficient than all three exact methods across all regimes in terms of the leading dimension dependence.

While the gain in efficiency of Compressed Scoring comes at the price of approximation, empirically, we find the approximate scores remain a reliable surrogate for ranking or filtering training samples [Hu et al., 2025a]. In our implementation, we therefore use Compressed Scoring as the default choice.

4.4 Compatibility with Memory-Saving Techniques in Modern Infrastructure

Modern LLM training pipelines employ a range of memory-saving techniques to mitigate the tight memory constraints. Since our methods introduce additional per-sample computational dependencies beyond standard training, a key system question is whether the proposed methods remain compatible with these techniques, including parameter-efficient training, activation checkpointing, and gradient accumulation.

In this subsection, we show that the proposed methods, especially Layer-Wise Subset Update, can be largely integrated with common memory-saving strategies given our system optimizations.

4.4.1 Compatibility with Parameter-Efficient Training

We first consider two widely-used parameter-efficient training methods, LoRA [Hu et al., 2022] and MeSO [Zhao et al., 2024]. Both reduce memory consumption by restricting or compressing the trainable update space, and both are naturally compatible with our framework.

LoRA. Recall that LoRA reparametrizes each linear layer as $W_t^{(l)} + B_t^{(l)} A_t^{(l)}$, where $B_t^{(l)} \in \mathbb{R}^{\sqrt{d/L} \times r}$ and $A_t^{(l)} \in \mathbb{R}^{r \times \sqrt{d/L}}$ with rank $r \ll \sqrt{d/L}$, and only the adapter parameters are updated. Since $A_t^{(l)}$ and $B_t^{(l)}$ are themselves linear maps, per-sample gradients retain the same outer-product factorization as in the full-parameter case, with the adapter dimension $2r\sqrt{d/L}$ replacing the full dimension d/L. All scoring and gradient computation carry over directly at proportionally lower cost (details in Section C.5.1).

MeSO. MeSO methods reduce memory in a different way: instead of restricting the trainable parameters, they maintain optimizer states and updates in a compressed subspace. This is also compatible with our framework, since data regularization only changes how the update direction is *chosen*, not how the resulting gradient is *applied*. In particular, any scoring method from Section 4.3 can be used to determine the selected subset, after which the resulting gradient update is passed to the MeSO optimizer in the usual compressed form.

There is also a tighter integration between MeSO and the Compressed Scoring method in Section 4.3.3. Since both rely on low-dimensional gradient representations, the same compressed per-sample gradients can be used both for score computation and for the optimizer update, avoiding redundant computation. This makes MeSO especially well aligned with our framework: it reduces the memory cost of optimization itself, while also supporting efficient approximate scoring in the same compressed space. We detail the full space of design choices in Section C.5.2.

4.4.2 Compatibility with Memory-Saving Scheduling Techniques

We now turn to two techniques that reduce memory by changing the temporal structure of training: activation checkpointing and gradient accumulation. Unlike LoRA and MeSO, these methods affect *when* intermediate tensors are available, and therefore interact more directly with our tensor lifetime schedule.

Activation checkpointing. Activation checkpointing reduces memory by discarding some forward activations and recomputing them on demand during backward. Our one-pass tensor lifetime schedule is compatible with checkpointing whenever the parameter groups align with the checkpointing schedule, so that all computations associated with a group can be resolved within a checkpoint segment. Layer-Wise Subset Update satisfies this condition naturally, since each layer forms its own group and can therefore be resolved within its own backward step.

When a group spans multiple checkpoint segments, however, the one-pass schedule can no longer preserve all required local tensors across segment boundaries without defeating the purpose of checkpointing. In this case, one must revert to the two-pass implementation discussed in Section 4.2: the first pass computes scores and determines the subsets, and the second pass assembles the update (see Section C.2.2). Thus, the compatibility with checkpointing depends on whether the partition granularity is aligned with the memory-saving schedule.

Gradient accumulation. Gradient accumulation reduces peak memory by splitting a large batch into micro-batches that are processed sequentially, and the activations and pre-activation gradients associated with an earlier micro-batch are typically released before the next one is processed for saving memory. This conflicts with our one-pass implementation, which relies on retaining the pairs $(a_i^{(l)}, \partial \ell / \partial e_i^{(l)})$ for all samples in the full batch in order to compute scores and assemble the final update. When the rule for determining $S_{t,p}$ decomposes across micro-batches, however, this issue resolves: each micro-batch can be scored, solved, and assembled independently during its own forward and backward pass, and the resulting updates can then be accumulated across micro-batches. Thresholding is one such example. In contrast, rules such as top-k depend on statistics computed over the entire batch, so the micro-batches cannot be resolved independently. In this case, we again resort to the two-pass implementation described in Section C.2.2.

5 Experiments

We evaluate our proposed methods across three post-training paradigms: supervised fine-tuning (SFT; Section 5.1), reinforcement learning from human feedback (RLHF; Section 5.2), and reinforcement learning with verifiable rewards (RLVR; Section 5.3). Across these settings, we compare against state-of-the-art baselines and show consistent improvements in downstream-task performance and convergence speed. We then benchmark the efficiency of the end-to-end training pipeline and different scoring mechanisms in Section 5.4, demonstrating that our implementation incurs minimal overhead. Our code can be found at https://github.com/TRAIS-Lab/Dr.Post-Training.

Unless otherwise specified, all reported results are averaged over 5 random seeds with error bars given as standard errors of the mean, and evaluation is performed on held-out test sets disjoint from the data used for regularization or training. Additional details are deferred to Section D.

5.1 Supervised Fine-Tuning

We first evaluate our methods in the SFT setting, where a model is fine-tuned on an instruction-style training dataset and evaluated on a distinct downstream benchmark. This setting tests whether data-regularized updates can better transfer general supervision toward a target objective. Our results show that Layer-Wise Subset Update substantially outperforms the prior state-of-the-art online data selection baseline. We further provide a case study to analyze the factors driving this performance gap.

5.1.1 Setup

We fine-tune LLAMA-3.2-1B [Grattafiori et al., 2024] on four training/target pairs that span different task types and training-pool compositions. 1.) `alpaca` [Taori et al., 2023]/`samsum` [Gliwa et al., 2019] (dialogue summarization), 2.) `less-mix` [Xia et al., 2024a]/`tydiqa` [Clark et al., 2020] (multilingual extractive QA), 3.) `triviaqa` [Joshi et al., 2017]/`nq_open` [Kwiatkowski et al., 2019] (closed-book factoid QA), and 4.) `less-mix` [Xia et al., 2024a]/`squad` [Rajpurkar et al., 2016] (reading-comprehension QA without context).

Across all four settings we report results under LoRA, our default fine-tuning variant. To additionally probe how the choice of fine-tuning method interacts with data regularization and demonstrate flexibility of our method with different fine-tuning methods, we also report results for full-parameter fine-tuning and MeSO on `alpaca/samsum`. All runs share the same fixed hyperparameters; see Section D.2 for the full configuration.

Data regularization is done with a small held-out target set (16 samples in total), following the setting of the prior state-of-the-art online baseline GREATS [Wang et al., 2024b]. Specifically, we consider $n = 8$ and $m = 1$ for training, i.e., each optimization step has access to 8 samples from the general training set and 1 sample from the held-out target set. In our framework, GREATS corresponds to *Global Subset Update*, while our method uses *Layer-Wise Subset Update*, where we approximately solve Eq. (3) by the top-k among the scores with $k = n/2$.[5]

5.1.2 Results

Figure 6 shows the training dynamics of `alpaca/samsum` with three fine-tuning methods, and Table 2 reports downstream F1 on the other three QA settings, with Figure 7 showing their corresponding training dynamics. Layer-Wise Subset Update consistently and substantially outperforms both Full-Training Update and Global Subset Update on every QA setting (and Global Subset, in turn, improves on Full-Training). The gap is most pronounced when the training pool is broad, i.e., `less-mix/tydiqa` jumps from 10.1% to 27.4% F1 between Full-Training and Layer-Wise Subset, confirming that data regularization matters most when the general distribution is misaligned with the target. We defer a detailed efficiency analysis to Section 5.4.

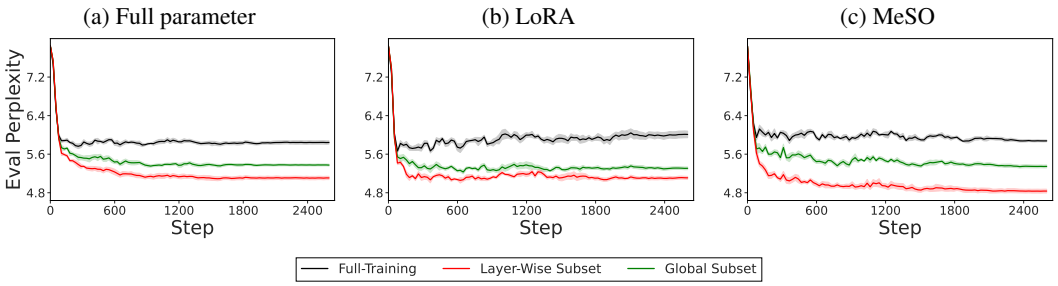

Figure 6: SFT training dynamics on `alpaca/samsum` across the three fine-tuning methods: evaluation perplexity throughout training.

[5]We omit the Target-Only Update in our experiments (i.e., training on only 16 samples) due to training instability.

Table 2: SFT downstream F1 (%) on the three QA settings.

Method	`less-mix/tydiqa`	`triviaqa/nq_open`	`less-mix/squad`
Full-Training	10.1 ± 0.2	12.6 ± 0.1	7.6 ± 0.1
Global Subset	13.4 ± 1.0	13.7 ± 0.2	8.3 ± 0.1
Layer-Wise Subset	$\mathbf{27.4 \pm 1.3}$	$\mathbf{14.6 \pm 0.4}$	$\mathbf{9.1 \pm 0.3}$

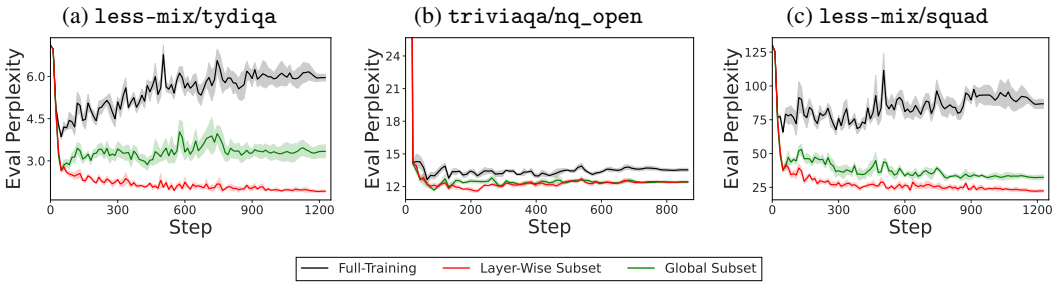

Figure 7: SFT training dynamics on the three QA settings: evaluation perplexity throughout training.

5.1.3 Case Study

To understand why Layer-Wise Subset Update outperforms Global Subset Update, we conduct a case study in which we record per-layer scores during Full-Training Update on `alpaca/samsum` with full-parameter fine-tuning. At each training step, we separately score the same batch of $n = 8$ samples as in Layer-Wise Subset Update and Global Subset Update, and compute (i) the mean absolute score per layer (magnitude), and (ii) the Spearman rank correlation between each layer's per-sample scores and the subset's global ranking. Figure 8 plots these quantities across all 16 blocks, with lines showing the mean across training steps and shaded bands the interquartile range. We aggregate the results by *transformer blocks* and *layer types*: each transformer block contains seven linear layers, namely four attention projections (query (Q), key (K), value (V), and output (O)) and three MLP projections (gate, up, and down).

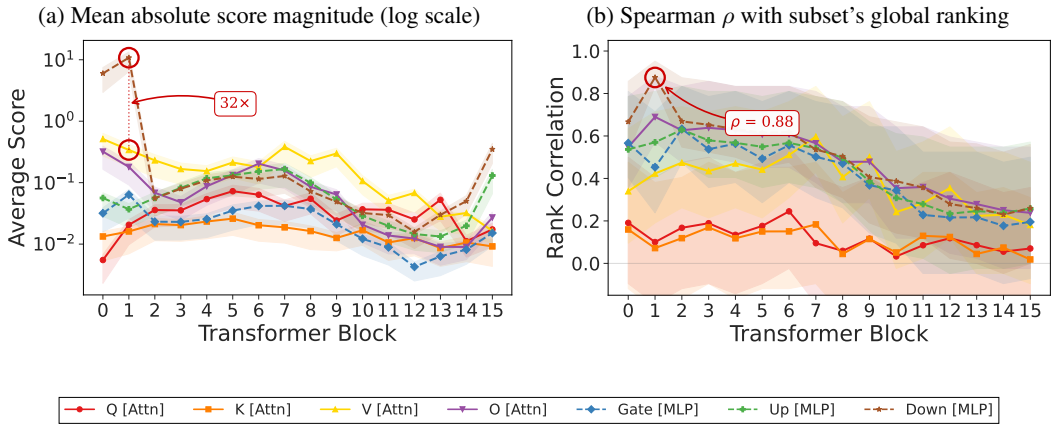

Figure 8: Per-layer scores on `alpaca/samsum`, averaged across training steps (lines) with 25^{th}–75^{th} interquartile bands. Score magnitudes vary by orders of magnitude across layer types, so Global Subset Update's global ranking is dominated by `down_proj` while layers like Q and K ($\rho \lesssim 0.2$) receive effectively uncurated data.

The magnitudes (Figure 8(a)) are heavily skewed: `down_proj` at block 1 produces scores $32\times$ larger than the second-largest layer type at the same block, even though K and V projections share an identical input–output shape (512×2048). Since Global Subset Update aggregates scores across all

layers into a single per-sample ranking, the resulting subset is dominated by whichever layer type has the largest magnitude. Figure 8(b) confirms this: `down_proj` achieves near-perfect agreement with the global ranking ($\rho \approx 0.88$), while Q and K projections remain near zero ($\rho \lesssim 0.2$). These 32 layers (out of 112 block-level linear layers) have essentially no influence on which samples they receive under Global Subset Update. Layer-Wise Subset Update resolves this by letting each layer independently determine its own training subset. The same pattern holds across the other three QA settings (Section D.2.4); the magnitude ratio between `down_proj` and the second-largest layer type ranges from $22\times$ to $38\times$, and the global-ranking correlation for the dominant `down_proj` layer ranges from $\rho \approx 0.90$ to $\rho \approx 0.94$.

5.2 Reinforcement Learning from Human Feedback

We next evaluate in the RLHF setting, where we apply Dr. Post-Training framework beyond the training-target distribution setting.

5.2.1 Setup

We follow a standard RLHF pipeline using `TRL` [von Werra et al., 2020] with PPO [Schulman et al., 2017] for detoxification [Hugging Face, 2023], using GPT-NEO-2.7B [Black et al., 2021] with LoRA. The policy generates continuations scored by a toxicity-based reward model (`LFTW R4 Target` [Vidgen et al., 2021]), and is evaluated using a separate toxicity detector (`da-electra-hatespeech-detection`). We compare four methods: (i) *standard* PPO without data curation, (ii) *IIF* [Hu et al., 2025b], which pre-filters the entire rollout batch once before PPO optimization begins (offline Global Subset Update), (iii) *Global Subset Update*, which curates each mini-batch during PPO with a single global selection across all layers, and (iv) *Layer-Wise Subset Update*, which independently selects samples per layer within each mini-batch.

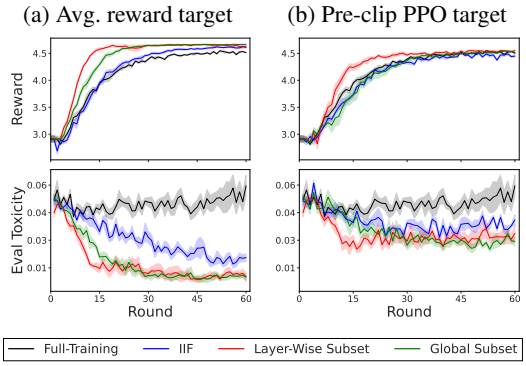

Figure 9: RLHF (*self-reference* target). **Top.** Training reward. **Bottom.** Evaluation toxicity.

All curation methods use negative score filtering, retaining only samples with non-negative gradient alignment. We further ablate two design choices: (i) *target signal source*: self-referencing (the same rollout batch) versus a new rollout on a held-out target set, and (ii) *target loss*: reward-weighted log-probability versus the pre–clip PPO surrogate. As described in Section 3.5, the PPO surrogate here acts as a stabilizing constraint on the (high-variance) policy gradient signal, generalizing the training-target distribution perspective.

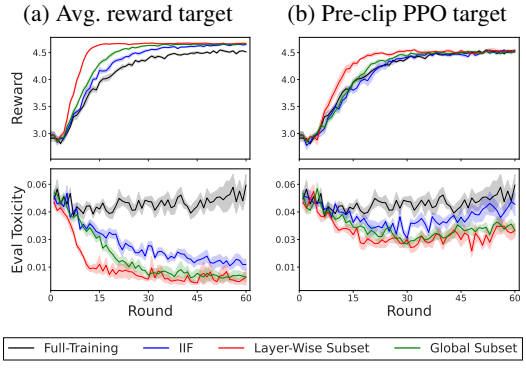

Figure 10: RLHF (*held-out* target). **Top.** Training reward. **Bottom.** Evaluation toxicity.

5.2.2 Results

Figures 9 and 10 show results across all four configurations. Layer-Wise Subset Update consistently achieves the highest training rewards and the lowest evaluation toxicity, followed by Global Subset Update, across both target signal sources and both target loss functions. Comparing the two target sources, the held-out target leads to higher final rewards, while the self-referencing target primarily accelerates early convergence. Among the target losses, the reward-weighted objective outperforms the pre-clip PPO surrogate in both settings.

5.3 Reinforcement Learning with Verifiable Rewards

Finally, we evaluate in the RLVR setting, where rewards are computed by an automatic verifier.

5.3.1 Setup

We use the `Verl` framework [Sheng et al., 2024] with GRPO [Shao et al., 2024] on the `MATH` dataset [Hendrycks et al., 2021] with QWEN3-1.7B [Qwen Team, 2025]. We compare Layer-Wise Subset Update against standard GRPO training (no data curation) and Global Subset Update. We instantiate the data regularization framework as in-run cross-validation: one mini-batch serves as the training batch, another as the target batch, and gradient alignment scores drive the negative score filtering of the training batch.

5.3.2 Results

Figure 11 shows our proposed Layer-Wise Subset Update method again achieves the best downstream performance in the RLVR setting. In particular, Global Subset Update accelerates early training but converges to roughly the same accuracy as standard GRPO. Layer-Wise Subset Update, by contrast, sustains its advantage throughout and achieves a consistently higher final accuracy.

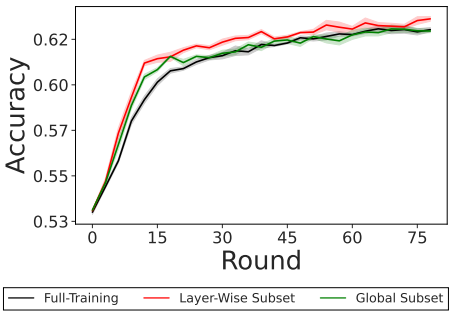

Figure 11: RLVR on `MATH` with QWEN3-1.7B. Evaluation accuracy over training.

5.4 System Efficiency

We benchmark the system efficiency of different methods in Section 3 and scoring mechanisms from Section 4: (i) *Breakdown*: end-to-end training overhead at $n = 8, T = 512, m = 1$ with compressed scoring; and (ii) *Scoring*: standalone timing on synthetic tensors at fixed $n = 8$, sweeping T at $m = 1$ and m at $T = 512$ to validate the crossover predictions from Section 4.3. All benchmarks are conducted on three models from three different architecture families—SMOLLM2-360M [Allal et al., 2025], TINYLLAMA-1.1B [Zhang et al., 2024], and LLAMA-3.2-3B [Grattafiori et al., 2024]—using `bfloat16` on a single `A40` GPU. The three families are chosen to factor out family-specific kernel quirks.

5.4.1 Per-Step Breakdown

Table 3 reports the wall-clock overhead of each method relative to Full-Training Update, where each column is averaged over 20 timed iterations after 10 warmup iterations.

The backward pass is broken into four components: `a.grad` computes the input activation gradient $\partial \ell / \partial a^{(l)}$ via a single matmul with the weight matrix (the output gradient $\partial \ell / \partial e^{(l)}$ is propagated from the layer above by autograd and requires no computation at this layer); `scoring` computes per-sample alignment scores; `w.grad` assembles the weight gradient from the selected samples; and `autograd` captures the remaining overhead: autograd graph traversal, memory allocation, hook dispatch, and backward passes of non-linear modules (LayerNorm, activation functions, etc.).

Table 3: Per-step component breakdown (ms) with standard training (Algorithm 4.1), using compressed scoring ($n = 8, T = 512, m = 1, k = 4$).

	SMOLLM2-360M			TINYLLAMA-1.1B			LLAMA-3.2-3B		
Component	Full-Training	Layer-Wise Subset	Global Subset	Full-Training	Layer-Wise Subset	Global Subset	Full-Training	Layer-Wise Subset	Global Subset
Forward	78.4	88.6	88.9	155.5	168.4	167.9	343.3	387.5	390.0
Backward	155.7	234.0	193.0	312.5	352.8	344.2	828.2	825.6	820.2
`a.grad`	32.5	34.2	33.3	93.3	96.6	96.6	228.9	255.8	256.8
`scoring`	—	43.8	22.5	—	35.6	31.3	—	49.7	45.4
`w.grad`	27.5	43.5	30.7	91.7	77.2	73.4	342.2	233.1	235.4
`autograd`	95.6	112.7	106.5	127.6	143.4	142.9	257.1	287.1	282.6
Optimizer	26.6	26.6	26.6	81.2	81.3	81.2	235.4	235.4	235.5
Total	**261**	**349** (+34%)	**309** (+18%)	**549**	**602** (+10%)	**593** (+8%)	**1407**	**1449** (+3%)	**1446** (+3%)

Results. We see that both Layer-Wise and Global Subset Update add only 3–34% wall-clock overhead compared to Full-Training Update, with overhead shrinking sharply as the model grows: from 34% (Layer-Wise) and 18% (Global Subset) on SMOLLM2-360M down to 3% for both methods on LLAMA-3.2-3B. Focusing on `w.grad`, we see that when the model is small (SMOLLM2-360M), the time required for both Layer-Wise and Global Subset is similar to or even larger than Full-Training Update, despite the fact that we only materialize the batch gradient from $k = 4$ samples. This is mainly due to the artifact of the model being too small: as the model grows, the differences start to show, eventually approaching $n/k = 2$ times more efficient when the bottlenecks become compute-bound rather than IO/kernel launching-bound (e.g., on LLAMA-3.2-3B `w.grad` drops from 342 ms to $\sim$234 ms, a $\sim$1.46$\times$ reduction). This further explains why the overheads become smaller as the model size grows.

Activation checkpointing. As we discussed in Section 4.4, the key efficiency difference between different partition granularities under Group-Wise Subset Update, such as Global Subset Update and Layer-Wise Subset Update, is their memory footprint and compatibility with modern memory-saving techniques like activation checkpointing.

Table 4: Per-step component breakdown (ms) with activation checkpointing, using compressed scoring ($n = 8, T = 512, m = 1, k = 4$).

	SMOLLM2-360M			TINYLLAMA-1.1B			LLAMA-3.2-3B		
Component	Full-Training	Layer-Wise Subset	Global Subset	Full-Training	Layer-Wise Subset	Global Subset	Full-Training	Layer-Wise Subset	Global Subset
Forward	88.2	109.7	107.7	154.9	167.4	167.1	389.2	437.4	438.4
Backward	249.0	311.4	297.1	453.6	504.3	495.9	1233.5	1279.1	1267.8
`a.grad`	40.7	41.0	40.8	93.3	96.8	96.7	269.4	299.4	299.3
`scoring`	—	33.4	26.6	—	35.5	31.3	—	60.2	54.7
`w.grad`	36.2	43.3	37.5	91.8	77.1	73.4	371.6	254.9	254.0
`autograd`	172.0	193.8	192.1	268.5	294.9	294.5	592.5	664.6	659.7
Optimizer	26.8	26.8	26.8	81.2	81.3	81.3	236.8	236.8	236.8
Total	**364**	**448** (+23%)	**432** (+19%)	**690**	**753** (+9%)	**744** (+8%)	**1860**	**1953** (+5%)	**1943** (+4%)

Tables 3 and 4 illustrate this clearly. From Table 4, we see that the relative computational overheads remain in the same range (4–19% for Global Subset Update, 5–23% for Layer-Wise Subset Update) as those without checkpointing (Table 3), since checkpointing inflates both the baseline and the scoring-augmented backward pass roughly in proportion as we expected.

On the other hand, Table 5 reports peak GPU memory. Here, we see that without checkpointing, peak memory is nearly identical across all methods at the same m; the overhead relative to Full-Training Update is determined almost entirely by the m additional target samples in the merged batch. With checkpointing enabled, the memory overhead of Global Subset Update becomes apparent: as discussed in Section 4.4, for a partition that spans a large number of layers across the checkpointing schedule, the designed one-pass scheduling fails to respect it, resulting in a similar peak memory usage as without checkpointing when the model size grows.

Table 5: Peak GPU memory (GB) using compressed scoring ($n = 8, T = 512, m = 1, k = 4$), with and without activation checkpointing.

	SMOLLM2-360M			TINYLLAMA-1.1B			LLAMA-3.2-3B		
	Full-Training	Layer-Wise Subset	Global Subset	Full-Training	Layer-Wise Subset	Global Subset	Full-Training	Layer-Wise Subset	Global Subset
No ckpt	9.4	10.3	10.4	15.0	16.2	16.2	37.9	40.7	40.7
With ckpt	4.6	4.9	7.5	10.3	10.4	14.5	29.9	30.1	38.1

5.4.2 Scoring Cost

We next benchmark different scoring mechanisms introduced in Section 4.3. Table 6 reports standalone scoring cost at $n = 8$, sweeping T at fixed $m = 1$ and m at fixed $T = 512$.

Results. We see that across the model size, number of validation samples (m), and also the length of the sequence (T), our proposed PIP remains competitive across regimes. More specifically, this validates the theoretical crossovers from Section 4.3.

At short sequences ($T = 512$), GIP is the cheapest exact method for all three models (42–71 ms). At $T = 2048$, the ranking reverses for models with smaller $\sqrt{d/L}$: Direct becomes cheapest on SMOLLM2-360M (151 vs. GIP 441 ms) and TINYLLAMA-1.1B (349 vs. 518 ms), while LLAMA-3.2-3B (whose larger $\sqrt{d/L}$ pushes the crossover beyond $T = 2048$) retains GIP as cheapest (946

Table 6: Standalone scoring cost (ms/step) at $n = 8$. Bold indicates the cheapest exact method per column. The T-sweep fixes $m = 1$; the m-sweep fixes $T = 512$.

	SmolLM2-360M						TinyLlama-1.1B						Llama-3.2-3B					
	sweep T			sweep m			sweep T			sweep m			sweep T			sweep m		
Scoring	512	2048	8192	1	4	8	512	2048	8192	1	4	8	512	2048	8192	1	4	8
Direct	56	**151**	**571**	56	77	99	131	**349**	**1344**	131	174	234	415	1166	**4076**	415	518	668
GIP	**42**	441	7073	**42**	130	239	**43**	518	8317	**43**	153	275	**71**	**946**	15017	**71**	**273**	495
PIP	53	155	577	53	**71**	**87**	103	393	1567	103	**141**	**189**	254	1076	4127	254	344	**465**
Compressed	29	48	162	29	29	29	25	58	216	25	25	25	40	107	412	40	40	40

ms). By $T = 8192$, all models have crossed over and Direct is the cheapest exact method, with PIP as a close runner-up on SmolLM2-360M (571 vs. 577 ms) and Llama-3.2-3B (4076 vs. 4127 ms).

The m-dependent crossover is also confirmed: at $T = 512$, PIP overtakes GIP at $m = 4$ for SmolLM2-360M (71 vs. 130 ms) and TinyLlama-1.1B (141 vs. 153 ms), and at $m = 8$ for Llama-3.2-3B (465 vs. 495 ms), consistent with the predicted crossover at $m > \sqrt{d/L}/(2T)$. Compressed scoring remains cheapest across all regimes (25–412 ms), making it the practical default.

6 Discussion

In this section, we discuss several aspects related to our proposed framework.

6.1 Classical Regularization

Both data regularization and classical regularization (e.g., weight decay, KL penalties, LoRA's low-rank constraint) ultimately constrain the parameter update, but they differ in the *source* of the constraint. Classical regularization is usually derived from a *data-agnostic* complexity measure on θ, restricting the update through intrinsic properties of the parameters themselves. Data regularization, by contrast, imposes a *data-induced* constraint in the parameter-update space via the feasible set U_t, whose admissible directions are shaped by the available training samples.

6.2 Designing New Feasible Sets

The Dr. Post-Training framework provides a blueprint for designing new data-centric methods: (i) define a feasible set U, (ii) design an efficient algorithm that solves $u^* \in U$ based on Eq. (3). Several natural extensions of the feasible set follow this template:

1. **Soft weighting.** Replace hard subset constraints with continuous weights: $U_{\text{soft}} = \{\sum_i w_i g_i : w \in \Delta_n\}$, where Δ_n is the probability simplex. The projection Eq. (3) becomes a quadratic program over the simplex, solvable in closed form or via efficient Frank-Wolfe steps [Nikdan et al., 2025]. This can be further extended to the same group-wise decomposition across parameter groups. This relaxation sidesteps the combinatorial search over $\binom{n}{k}$ subsets required previously.
2. **Token-level granularity.** For sequential models, instead of operating at the sample level, operate at the token level within each sample, yielding a finer-grained feasible set. Since per-token gradients are naturally available during backpropagation, this can be implemented within the existing Group-Wise Subset Update pipeline.

6.3 Connection to Domain Adaptation

The mismatch between general-purpose pretraining corpora and a narrow target task is a recurring theme in transfer learning and domain adaptation [Pan and Yang, 2009]. A long line of work has studied when and how such transfer succeeds, highlighting source–target divergence and the risk of negative transfer as central concerns [Ben-David et al., 2010]. Classical remedies operate at the instance level: reweighting examples under covariate shift [Sugiyama et al., 2007], adapting feature representations [Daumé III, 2007], aligning domains adversarially [Ganin et al., 2016], or combining multiple source domains [Guo et al., 2018]. In the LLM era, continued in-domain or task-adaptive pretraining has become the de facto approach for reducing this mismatch [Gururangan et al., 2020].

Our framework addresses the same general training–target gap, but at a fundamentally different level of abstraction. Where instance reweighting corrects the empirical risk via density ratios and representation alignment enforces domain invariance in hidden states, both are specific algorithmic choices for bridging the domain gap. Dr. Post-Training instead formulates general training data as a *regularizer* on the target-driven update direction, yielding an explicit per-step bias–variance tradeoff governed by the feasible set U. This perspective not only subsumes existing instance-level strategies as special cases of the feasible set but also opens the door to *beyond-instance* designs such as Group-Wise Subset Update, where different parameter groups draw on different training subsets within a single optimization step—a capability that falls outside the scope of standard domain-adaptation formulations.

7 Conclusion

In this work, we propose **Dr. Post-Training**, a data regularization framework for LLM post-training that leverage the abundant yet imperfectly aligned general training data a regularizer that constrains target-driven updates. This framework provides an alternative view of existing data selection methods and further leads to natural generalization, broadening the design space and shedding light on their respective tradeoffs. With extensive system optimization, we offer an efficient implementation of our proposed methods under strict memory constraints in modern LLM training pipelines. Experiments across SFT, RLHF, and RLVR demonstrated consistent improvements over strong baselines with minimal system overhead.

Acknowledgment

The authors would like to thank Joe Melkonian for helpful discussions at the early stage of this project. P. Hu and J.W. Ma are partially supported by a gift grant from Google.

References

A. Albalak, Y. Elazar, S. M. Xie, S. Longpre, N. Lambert, X. Wang, N. Muennighoff, B. Hou, L. Pan, H. Jeong, C. Raffel, S. Chang, T. Hashimoto, and W. Y. Wang. A survey on data selection for language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=XfHWcNTSHp. Survey Certification.

L. B. Allal, A. Lozhkov, E. Bakouch, G. M. Blázquez, G. Penedo, L. Tunstall, A. Marafioti, H. Kydlíček, A. P. Lajarín, V. Srivastav, et al. Smollm2: When smol goes big–data-centric training of a small language model. *arXiv preprint arXiv:2502.02737*, 2025.

J. L. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

S. Ben-David, J. Blitzer, K. Crammer, A. Kulesza, F. Pereira, and J. W. Vaughan. A theory of learning from different domains. *Machine learning*, 79(1):151–175, 2010.

S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow, Mar. 2021. URL https://doi.org/10.5281/zenodo.5297715. If you use this software, please cite it using these metadata.

L. Bottou, F. E. Curtis, and J. Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

Y. Cao, Y. Kang, C. Wang, and L. Sun. Instruction mining: Instruction data selection for tuning large language models. *arXiv preprint arXiv:2307.06290*, 2023.

S. Casper, X. Davies, C. Shi, T. K. Gilbert, J. Scheurer, J. Rando, R. Freedman, T. Korbak, D. Lindner, P. Freire, et al. Open problems and fundamental limitations of reinforcement learning from human feedback. *arXiv preprint arXiv:2307.15217*, 2023.

L. Chen, S. Li, J. Yan, H. Wang, K. Gunaratna, V. Yadav, Z. Tang, V. Srinivasan, T. Zhou, H. Huang, et al. Alpagasus: Training a better alpaca with fewer data. In *The Twelfth International Conference on Learning Representations*, 2024.

T. Chen, B. Xu, C. Zhang, and C. Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

S. K. Choe, H. Ahn, J. Bae, K. Zhao, M. Kang, Y. Chung, A. Pratapa, W. Neiswanger, E. Strubell, T. Mitamura, et al. What is your data worth to gpt? llm-scale data valuation with influence functions. In *Advances in Neural Information Processing Systems*, 2025.

P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

J. H. Clark, E. Choi, M. Collins, D. Garrette, T. Kwiatkowski, V. Nikolaev, and J. Palomaki. Tydi qa: A benchmark for information-seeking question answering in ty pologically di verse languages. *Transactions of the Association for Computational Linguistics*, 8:454–470, 2020.

H. Daumé III. Frustratingly easy domain adaptation. In *Proceedings of the 45th annual meeting of the association of computational linguistics*, pages 256–263, 2007.

J. Deng, Y. Hu, P. Hu, T.-W. Li, S. Liu, J. T. Wang, D. Ley, Q. Dai, B. Huang, J. Huang, C. Jiao, H. A. Just, Y. Pan, J. Shen, Y. Tu, W. Wang, X. Wang, S. Zhang, S. Zhang, R. Jia, H. Lakkaraju, H. Peng, W. Tang, C. Xiong, J. Zhao, H. Tong, H. Zhao, and J. W. Ma. A survey of data attribution: Methods, applications, and evaluation in the era of generative ai. *SSRN*, 2025. doi: 10.2139/ssrn.5451054. Available at SSRN: https://ssrn.com/abstract=5451054.

T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, 36:10088–10115, 2023.

K. Ethayarajh, Y. Choi, and S. Swayamdipta. Understanding dataset difficulty with $\mathcal{V}$-usable information. In *International Conference on Machine Learning*, pages 5988–6008. PMLR, 2022.

Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. March, and V. Lempitsky. Domain-adversarial training of neural networks. *Journal of machine learning research*, 17 (59):1–35, 2016.

S. Gehman, S. Gururangan, M. Sap, Y. Choi, and N. A. Smith. Realtoxicityprompts: Evaluating neural toxic degeneration in language models. *arXiv preprint arXiv:2009.11462*, 2020.

A. Ghorbani and J. Zou. Data shapley: Equitable valuation of data for machine learning. In *International conference on machine learning*, pages 2242–2251. PMLR, 2019.

B. Gliwa, I. Mochol, M. Biesek, and A. Wawer. Samsum corpus: A human-annotated dialogue dataset for abstractive summarization. *EMNLP-IJCNLP 2019*, page 70, 2019.

A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

S. Gunasekar, Y. Zhang, J. Aneja, C. C. T. Mendes, A. Del Giorno, S. Gopi, M. Javaheripi, P. Kauffmann, G. de Rosa, O. Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

J. Guo, D. Shah, and R. Barzilay. Multi-source domain adaptation with mixture of experts. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 4694–4703, 2018.

S. Gururangan, A. Marasović, S. Swayamdipta, K. Lo, I. Beltagy, D. Downey, and N. A. Smith. Don't stop pretraining: Adapt language models to domains and tasks. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 8342–8360, 2020.

X. Han and Y. Tsvetkov. Influence tuning: Demoting spurious correlations via instance attribution and instance-driven update. In M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4398–4409, Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.374. URL https://aclanthology.org/2021.findings-emnlp.374/.

Z. Han, C. Gao, J. Liu, J. Zhang, and S. Q. Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=lIsCS8b6zj.

T. Hastie, R. Tibshirani, J. H. Friedman, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

E. Hazan. Introduction to online convex optimization. *Foundations and Trends in Optimization*, 2 (3-4):157–325, 2016.

L. He, M. Xia, and P. Henderson. What is in your safe data? identifying benign data that breaks safety. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=Hi8jKh4HE9.

Y. He, P. Li, Y. Hu, C. Chen, and K. Yuan. Subspace optimization for large language models with convergence guarantees. In *Forty-second International Conference on Machine Learning*, 2025.

D. Hendrycks, C. Burns, S. Kadavath, A. Arora, S. Basart, E. Tang, D. Song, and J. Steinhardt. Measuring mathematical problem solving with the math dataset. *NeurIPS*, 2021.

E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

P. Hu, J. Melkonian, W. Tang, H. Zhao, and J. W. Ma. GraSS: Scalable data attribution with gradient sparsification and sparse projection. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025a. URL https://openreview.net/forum?id=o0HgWRmyY1.

Y. Hu, P. Hu, H. Zhao, and J. Ma. Most influential subset selection: Challenges, promises, and beyond. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 119778–119810. Curran Associates, Inc., 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/d8684e49752e06ac5e4b554b60ad212a-Paper-Conference.pdf.

Y. Hu, F. Wu, H. Ye, D. Forsyth, J. Zou, N. Jiang, J. W. Ma, and H. Zhao. A snapshot of influence: A local data attribution framework for online reinforcement learning. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025b. URL https://openreview.net/forum?id=sYK4yPDuT1.

Hugging Face. Detoxifying a language model using ppo. https://huggingface.co/docs/trl/en/detoxifying_a_lm, 2023. TRL documentation (v0.17.0), accessed May 8, 2025.

M. F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 18(3):1059–1076, 1989.

H. Ivison, N. A. Smith, H. Hajishirzi, and P. Dasigi. Data-efficient finetuning using cross-task nearest neighbors. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 9036–9061, 2023.

S. Iyer, X. V. Lin, R. Pasunuru, T. Mihaylov, D. Simig, P. Yu, K. Shuster, T. Wang, Q. Liu, P. S. Koura, et al. Opt-iml: Scaling language model instruction meta learning through the lens of generalization. *arXiv preprint arXiv:2212.12017*, 2022.

C. Jiao, W. Gao, A. Raghunathan, and C. Xiong. On the feasibility of in-context probing for data attribution. In L. Chiruzzo, A. Ritter, and L. Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 5140–5155, Albuquerque, New Mexico, Apr. 2025. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.286. URL https://aclanthology.org/2025.findings-naacl.286/.

M. Joshi, E. Choi, D. S. Weld, and L. Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1601–1611, 2017.

P. W. Koh and P. Liang. Understanding black-box predictions via influence functions. In *International conference on machine learning*, pages 1885–1894. PMLR, 2017.

P.-N. Kung, F. Yin, D. Wu, K.-W. Chang, and N. Peng. Active instruction tuning: Improving cross-task generalization by training on prompt sensitive tasks. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 1813–1829, 2023.

T. Kwiatkowski, J. Palomaki, O. Redfield, M. Collins, A. Parikh, C. Alberti, D. Epstein, I. Polosukhin, J. Devlin, K. Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

N. Lambert, J. Morrison, V. Pyatkin, S. Huang, H. Ivison, F. Brahman, L. J. V. Miranda, A. Liu, N. Dziri, S. Lyu, Y. Gu, S. Malik, V. Graf, J. D. Hwang, J. Yang, R. L. Bras, O. Tafjord, C. Wilhelm, L. Soldaini, N. A. Smith, Y. Wang, P. Dasigi, and H. Hajishirzi. Tülu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

G. Lan. *First-order and stochastic optimization methods for machine learning*, volume 1. Springer, 2020.

K. Lange. *MM optimization algorithms*. SIAM, 2016.

K. Lange, D. R. Hunter, and I. Yang. Optimization transfer using surrogate objective functions. *Journal of computational and graphical statistics*, 9(1):1–20, 2000.

C. Ling, X. Zhao, J. Lu, C. Deng, C. Zheng, J. Wang, T. Chowdhury, Y. Li, H. Cui, X. Zhang, et al. Domain specialization as the key to make large language models disruptive: A comprehensive survey. *ACM Computing Surveys*, 58(3):1–39, 2025.

Z. Liu, T. D. Nguyen, T. H. Nguyen, A. Ene, and H. Nguyen. High probability convergence of stochastic gradient methods. In *International conference on machine learning*, pages 21884–21914. PMLR, 2023.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

K. Lu, H. Yuan, Z. Yuan, R. Lin, J. Lin, C. Tan, C. Zhou, and J. Zhou. #instag: Instruction tagging for analyzing supervised fine-tuning of large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pszewhybU9.

J. Mairal. Optimization with first-order surrogate functions. In *International conference on machine learning*, pages 783–791. PMLR, 2013.

S. Mishra, D. Khashabi, C. Baral, and H. Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3470–3487, 2022.

N. Muennighoff, S. Hongjin, L. Wang, N. Yang, F. Wei, T. Yu, A. Singh, and D. Kiela. Generative representational instruction tuning. In *The Thirteenth International Conference on Learning Representations*, 2024.

A. Muhamed, O. Li, D. Woodruff, M. Diab, and V. Smith. Grass: Compute efficient low-memory llm training with structured sparse gradients. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 14978–15003, 2024.

Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

M. Nikdan, V. Cohen-Addad, D. Alistarh, and V. Mirrokni. Efficient data selection at scale via influence distillation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=E6ZdfjtoiX.

L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

S. J. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2009.

S. Pandya, P. Patel, B. D. Nord, M. Walmsley, and A. Ćiprijanović. Sidda: Sinkhorn dynamic domain adaptation for image classification with equivariant neural networks. *Machine Learning: Science and Technology*, 6(3):035032, 2025.

E. Perez, D. Kiela, and K. Cho. True few-shot learning with language models. *Advances in neural information processing systems*, 34:11054–11070, 2021.

G. Pruthi, F. Liu, S. Kale, and M. Sundararajan. Estimating training data influence by tracing gradient descent. *Advances in Neural Information Processing Systems*, 33:19920–19930, 2020.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 conference on empirical methods in natural language processing*, pages 2383–2392, 2016.

J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506, 2020.

A. Renduchintala, T. Konuk, and O. Kuchaiev. Tied-lora: Enhancing parameter efficiency of lora with weight tying. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 8694–8705, 2024.

J. Schulman and T. M. Lab. Lora without regret. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250929. https://thinkingmachines.ai/blog/lora/.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Y. Sheng, S. Cao, D. Li, C. Hooper, N. Lee, S. Yang, C. Chou, B. Zhu, L. Zheng, K. Keutzer, J. E. Gonzalez, and I. Stoica. S-lora: Serving thousands of concurrent lora adapters. *arXiv preprint arXiv:2311.03285*, 2023.

M. Sugiyama, M. Krauledat, and K.-R. Müller. Covariate shift adaptation by importance weighted cross validation. *Journal of Machine Learning Research*, 8(5), 2007.

R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

R. Vershynin. *High-Dimensional Probability: An Introduction with Applications in Data Science*. Cambridge University Press, 2018.

B. Vidgen, T. Thrush, Z. Waseem, and D. Kiela. Learning from the worst: Dynamically generated datasets to improve online hate detection. In *ACL*, 2021.

L. von Werra, Y. Belkada, L. Tunstall, E. Beeching, T. Thrush, N. Lambert, S. Huang, K. Rasul, and Q. Gallouédec. Trl: Transformer reinforcement learning. https://github.com/huggingface/trl, 2020.

J. Wang, X. Lin, R. Qiao, C.-S. Foo, and B. K. H. Low. Helpful or harmful data? fine-tuning-free shapley attribution for explaining language model predictions. In *Forty-first International Conference on Machine Learning*, 2024a. URL https://openreview.net/forum?id=WSpPC1Jm0p.

J. T. Wang, T. Wu, D. Song, P. Mittal, and R. Jia. Greats: Online selection of high-quality data for llm training in every iteration. *Advances in Neural Information Processing Systems*, 37: 131197–131223, 2024b.

Y. Wang, H. Ivison, P. Dasigi, J. Hessel, T. Khot, K. Chandu, D. Wadden, K. MacMillan, N. A. Smith, I. Beltagy, et al. How far can camels go? exploring the state of instruction tuning on open resources. *Advances in Neural Information Processing Systems*, 36:74764–74786, 2023a.

Y. Wang, Y. Lin, X. Zeng, and G. Zhang. Multilora: Democratizing lora for better multi-task learning. *arXiv preprint arXiv:2311.11501*, 2023b.

J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=gEZrGCozdqR.

M. Xia, S. Malladi, S. Gururangan, S. Arora, and D. Chen. Less: Selecting influential data for targeted instruction tuning. In *International Conference on Machine Learning*, pages 54104–54132. PMLR, 2024a.

W. Xia, C. Qin, and E. Hazan. Chain of lora: Efficient fine-tuning of language models via residual learning. *arXiv preprint arXiv:2401.04151*, 2024b.

B. Zhang and R. Sennrich. Root mean square layer normalization. *Advances in neural information processing systems*, 32, 2019.

L. Zhang, L. Zhang, S. Shi, X. Chu, and B. Li. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *arXiv preprint arXiv:2308.03303*, 2023.

P. Zhang, G. Zeng, T. Wang, and W. Lu. Tinyllama: An open-source small language model, 2024.

J. Zhao, Z. Zhang, B. Chen, Z. Wang, A. Anandkumar, and Y. Tian. Galore: Memory-efficient llm training by gradient low-rank projection. In *International Conference on Machine Learning*, pages 61121–61143. PMLR, 2024.

C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. YU, S. Zhang, G. Ghosh, M. Lewis, L. Zettlemoyer, and O. Levy. LIMA: Less is more for alignment. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=KBMOKmX2he.

A Omitted Details from Section 2

In this section, we briefly review the two low-dimensional parameter training methods used in this paper, LoRA and MeSO.

To keep the discussion self-contained, we introduce only the notation needed here. In particular, for a linear layer l, we write its weight matrix as $W^{(l)} \in \mathbb{R}^{d_{\text{out}}^{(l)} \times d_{\text{in}}^{(l)}}$, where $d_{\text{in}}^{(l)}$ and $d_{\text{out}}^{(l)}$ denote the input and output dimensions, respectively. This notation is consistent with the main text, except that here we do not assume $d_{\text{in}}^{(l)} = d_{\text{out}}^{(l)} = \sqrt{d/L}$.

A.1 LoRA

LoRA [Hu et al., 2022] reparameterizes each linear layer l as $W^{(l)} + B^{(l)}A^{(l)}$, where $B^{(l)} \in \mathbb{R}^{d_{\text{out}}^{(l)} \times r}$ and $A^{(l)} \in \mathbb{R}^{r \times d_{\text{in}}^{(l)}}$ are low-rank adapter matrices with *rank* $r \ll \min(d_{\text{in}}^{(l)}, d_{\text{out}}^{(l)})$. During training, the pretrained weight $W^{(l)}$ is kept frozen, and only the adapter parameters $A^{(l)}$ and $B^{(l)}$ are updated. We collectively refer to these trainable parameters as the *adapters*. Because the number of trainable parameters is much smaller than in full fine-tuning, LoRA substantially reduces memory usage: gradients and optimizer states (which can be several times larger than the gradients themselves) need to be stored only for the adapters rather than for the full weight matrix.

A.2 MeSO

MeSO [Zhao et al., 2024] maintains optimizer states in a compressed subspace $\mathbb{R}^{\kappa}$ rather than the full parameter space $\mathbb{R}^{d}$. At each layer l, MeSO uses a projection $\Pi \in \mathbb{R}^{\kappa^{(l)} \times d^{(l)}}$ (where $d^{(l)} = d_{\text{in}}^{(l)} d_{\text{out}}^{(l)}$ is the number of parameters, $\kappa^{(l)}$ is the compressed dimension for layer l, with $d = \sum_l d^{(l)}$ and $\kappa = \sum_l \kappa^{(l)}$) to compress the vectorized gradient $\text{vec}(g^{(l)})$. The standard MeSO update is given in Algorithm A.1.

Algorithm A.1: Standard MeSO Training

Data: Model θ_t, training data $\{z_i\}_{i=1}^n$, learning rate η_t, projections Π
Result: Updated model parameters θ_{t+1}

$(\ell, \{a_i^{(l)}, e_i^{(l)}\}_{l,i}) \leftarrow$ `Forward(`θ_t, $\{z_i\}_{i=1}^n$`)`
for $l = L, \ldots, 1$ **do**
  $\{\partial \ell / \partial e_i^{(l)}\}_{i=1}^n \leftarrow$ `Backward(`ℓ, $\theta_t^{(l)}$`)`
  $u_t^{(l)} \leftarrow \frac{1}{n} \sum_{i=1}^n (\partial \ell / \partial e_i^{(l)}) \otimes a_i^{(l)}$
  `Release(`$a_i^{(l)}$, $\partial \ell / \partial e_i^{(l)}$`)` for all i
  $\widetilde{u}_t^{(l)} \leftarrow \Pi(u_t^{(l)})$ `// Compress to` $\mathbb{R}^{\kappa^{(l)}}$
  $\theta_{t+1}^{(l)} \leftarrow \theta_t^{(l)} - \eta_t \cdot (\Pi^\top \widetilde{u}_t^{(l)})$ `// Back-project and update`
  `Release(`$\widetilde{u}_t^{(l)}$`)`
return θ_{t+1}

Compared to LoRA, MeSO achieves memory savings through a different mechanism. LoRA reduces memory by restricting training to a small set of adapter parameters, so gradients and optimizer states are stored only for those parameters. In contrast, MeSO keeps the original parameterization of the model, but performs optimization in a low-dimensional subspace. As a result, gradient-related quantities such as optimizer states need only be maintained in the compressed space rather than in the full parameter space. This can lead to even greater memory savings, especially when the subspace dimension is smaller than the number of trainable parameters used by LoRA.[6]

A.2.1 Subspace Refreshing

We note that in many MeSO methods, the projection Π is periodically refreshed from Π_{old} to Π_{new} to allow updating different subspaces, which requires transferring the optimizer's momentum states

[6] Technically, it is possible to combine LoRA and MeSO. However, this leads to instability and is not a common practice.

to the new subspace. Prior works omit this nuance by setting the momentum states to zero for simplicity [Zhao et al., 2024]. Here, we provide a strategy for AdamW [Loshchilov and Hutter, 2019] to mitigate this.

First moment. For the *first moment* $\hat{m}_{\text{old}} \in \mathbb{R}^{\kappa^{(l)}}$, the transfer is straightforward: back-project to full space via $\Pi_{\text{old}}^{\top}$, then forward-project via Π_{new}.

Second moment. For the *second moment* $\hat{v}_{\text{old}} \in \mathbb{R}^{\kappa^{(l)}}$, a naive linear transformation is incorrect since variance is a quadratic quantity. In our implementation we consider a diagonal covariance approximation $\hat{\Sigma}_{\text{old}} \approx \operatorname{diag}(\hat{v}_{\text{old}})$. This allows us to transform the second moment as:

$$\hat{v}_{\text{new}} = \operatorname{diag}(M \operatorname{diag}(\hat{v}_{\text{old}}) M^{\top}) = (M \odot M)\hat{v}_{\text{old}},$$

where $M = \Pi_{\text{new}}\Pi_{\text{old}}^{\top}$. When M is too large to form explicitly, this diagonal can be estimated stochastically via Hutchinson's method [Hutchinson, 1989]: $\hat{v}_{\text{new}} \approx \frac{1}{N_{\text{probe}}} \sum_{j=1}^{N_{\text{probe}}} r_j \odot (M \operatorname{diag}(\hat{v}_{\text{old}}) M^{\top} r_j)$ with Rademacher probes $r_j \in \{-1,1\}^{\kappa^{(l)}}$.

B Omitted Details from Section 3

This section collects the technical results deferred from Section 3. Section B.1 derives the projection formulation (Eq.(3)) from the majorization–minimization relaxation. Section B.2 derives both the one-step majorization descent lemma (Lemma 3.1) that reduces bias–variance tradeoffs for each method to MSE, and further provides the MSE bounds for all four methods, proving Proposition 3.3 and theorem 3.4.

For convenience, we restate all the assumptions used throughout the proofs.

Assumption (Target smoothness, Assumption 1)**.** *The target population loss $\mathcal{L}_\star$ is β-smooth for some $\beta > 0$.*

Assumption (Bounded gradients, Assumption 2)**.** *There exists $C > 0$ such that $\|g_i\| \le C$ for all $i \in [n]$.*

Assumption (Sub-Gaussian noise, Assumption 3)**.** *Noise of target gradient $\xi := \hat{g}_\star - g_\star$ is sub-Gaussian with parameter $\sigma/\sqrt{m}$ for some $\sigma > 0$.*

B.1 Derivation of the Projection Formulation

By Assumption 1, the target loss admits the quadratic majorization

$$\mathcal{L}_\star(\theta_t - \eta_t u) \le \mathcal{L}_\star(\theta_t) - \eta_t \langle g_\star, u\rangle + \frac{\beta\eta_t^2}{2}\|u\|^2.$$

Minimizing the upper bound on the right-hand side over $u \in U_t$, dropping the constant $\mathcal{L}_\star(\theta_t)$, and substituting the target batch estimate $\hat{g}_\star$ for the unknown $g_\star$ yields

$$u_t = \arg\max_{u \in U_t} \left\{ \langle \hat{g}_\star, u\rangle - \frac{\beta\eta_t}{2}\|u\|^2 \right\} \iff u_t \in \arg\min_{u \in U_t} \left\| u - \frac{\hat{g}_\star}{\beta\eta_t} \right\|^2,$$

where the equivalence follows from completing the square:

$$\langle \hat{g}_\star, u\rangle - \frac{\beta\eta_t}{2}\|u\|^2 = -\frac{\beta\eta_t}{2}\left\| u - \frac{1}{\beta\eta_t}\hat{g}_\star \right\|^2 + \frac{\|\hat{g}_\star\|^2}{2\beta\eta_t}$$

Setting $\eta_t = 1/\beta$ [Bottou et al., 2018] absorbs the scaling, giving Eq.(3).

B.2 Proof of the Bias–Variance Tradeoffs

We now prove the one-step majorization descent lemma (Lemma 3.1):

Lemma B.1. *Assume Assumption 1. Fix θ_t and let $0 < \eta_t \le 1/\beta$. For any u with $\mathbb{E}[\|u\|^2 \mid \theta_t] < \infty$,*

$$\mathbb{E}\left[\mathcal{L}_\star(\theta_{t+1}) \mid \theta_t\right] \le \mathcal{L}_\star(\theta_t) - \frac{\eta_t}{2}\|g_\star\|^2 + \frac{\eta_t}{2}\operatorname{MSE}(u).$$

Proof. By Assumption 1 and linearity of expectation,

$$\mathbb{E}[\mathcal{L}_\star(\theta_{t+1}) \mid \theta_t] \le \mathcal{L}_\star(\theta_t) - \eta_t \langle g_\star, \mathbb{E}[u \mid \theta_t] \rangle + \frac{\beta \eta_t^2}{2} \mathbb{E}[\|u\|^2 \mid \theta_t].$$

Expanding $\mathrm{MSE}(u) = \mathbb{E}[\|u\|^2 \mid \theta_t] - 2\langle g_\star, \mathbb{E}[u \mid \theta_t] \rangle + \|g_\star\|^2$ and rearranging gives

$$-\langle g_\star, \mathbb{E}[u \mid \theta_t] \rangle + \frac{1}{2} \mathbb{E}[\|u\|^2 \mid \theta_t] = -\frac{1}{2} \|g_\star\|^2 + \frac{1}{2} \mathrm{MSE}(u).$$

Substituting and using $\eta_t \le 1/\beta$ to drop the nonpositive remainder $(\frac{\beta \eta_t^2}{2} - \frac{\eta_t}{2}) \mathbb{E}[\|u\|^2 \mid \theta_t] \le 0$ yields the result. □

We are now ready to prove the bias–variance tradeoffs. Firstly, we prove Proposition 3.3 for the Full-Training Update and the Target-Only Update.

Proposition 1 (Bias–variance tradeoffs)**.** *Fix θ_t and let $\eta_t = 1/\beta$. Then:*

(i) ***Full-Training Update.*** *For $U_t^{\mathrm{tr}} = \{\hat{g}_{\mathrm{tr}}\}$,*

$$\mathcal{B}(U_t^{\mathrm{tr}}) = \|g_{\mathrm{tr}} - g_\star\|^2 + \frac{\mathrm{tr}(\Sigma_{\mathrm{tr}})}{n}, \qquad \mathcal{V}(u_t^{\mathrm{tr}}) = 0.$$

(ii) ***Target-Only Update.*** *For $U_t^\star = \mathbb{R}^d$,*

$$\mathcal{B}(U_t^\star) = 0, \qquad \mathcal{V}(u_t^\star) = \frac{\mathrm{tr}(\Sigma_\star)}{m}.$$

Proof. We prove this one by one.

(i) *Part (i) (Full-Training Update).* Since $u_t^{\mathrm{tr}} = \hat{g}_{\mathrm{tr}}$ has mean g_{tr} and covariance Σ_{tr}/n,

$$\mathrm{MSE}(u_t^{\mathrm{tr}}) = \|\mathbb{E}[\hat{g}_{\mathrm{tr}} - g_\star \mid \theta_t]\|^2 + \mathrm{tr}(\mathrm{Cov}(\hat{g}_{\mathrm{tr}} - g_\star \mid \theta_t)) = \|g_{\mathrm{tr}} - g_\star\|^2 + \frac{1}{n} \mathrm{tr}(\Sigma_{\mathrm{tr}}).$$

Furthermore, note that since $U_t^{\mathrm{tr}} = \{\hat{g}_t^{\mathrm{tr}}\}$, we also have $\mathcal{B}(U_t^{\mathrm{tr}}) = \mathrm{MSE}(u_t^{\mathrm{tr}})$.

(ii) *Part (ii) (Target-Only Update).* With $u_t^\star = \hat{g}_\star$ and $\hat{g}_\star - g_\star$ has mean zero and covariance $\Sigma_\star/m$,

$$\mathrm{MSE}(u_t^\star) = \mathbb{E}[\|\hat{g}_\star - g_\star\|^2 \mid \theta_t] = \mathrm{tr}(\mathrm{Cov}(\hat{g}_\star \mid \theta_t)) = \frac{1}{m} \mathrm{tr}(\Sigma_\star).$$

We conclude the proof by noting that $\mathcal{B}(U_t^\star) = 0$ as $U_t^\star = \mathbb{R}^d$.

□

Finally, we conclude this section by proving Theorem 3.4.

Theorem (Bias–variance tradeoffs)**.** *Assume Assumptions 2 and 3 and fix θ_t and let $\eta_t = 1/\beta$. Then for any $k \in [n]$:*

(i) ***Global Subset Update.*** *For $U_t^{\mathrm{glob}}(k)$,*

$$\mathcal{V}(u_t^{\mathrm{glob}}(k)) \le \frac{4C\sigma}{\sqrt{m}} \sqrt{2 \log\left(2\tbinom{n}{k}\right)}.$$

(ii) ***Group-Wise Subset Update.*** *For $U_t^{\mathrm{grp}}(k; \mathcal{G})$ where $\mathcal{G} = \{G_p\}_{p=1}^P$ contains P groups,*

$$\mathcal{V}(u_t^{\mathrm{grp}}(k; \mathcal{G})) \le \frac{4CP\sigma}{\sqrt{m}} \sqrt{2 \log\left(2\tbinom{n}{k}\right)}.$$

Moreover, Global Subset Update has a higher bias compared to Group-Wise Subset Update:

$$\mathcal{B}(U_t^{\mathrm{grp}}(k; \mathcal{G})) \le \mathcal{B}(U_t^{\mathrm{glob}}(k)).$$

Proof. We prove this one by one.

(i) *Part (i) (Global Subset Update).* Fix k and write $U \coloneqq U_t^{\mathrm{glob}}(k)$. Let $\xi \coloneqq \hat{g}_\star - g_\star$. Define

$$u^* \coloneqq u_t^{\mathrm{glob}} \in \arg\min_{u \in U} \|u - \hat{g}_\star\|^2, \qquad u^\dagger \in \arg\min_{u \in U} \|u - g_\star\|^2.$$

By the projection property, $\|u^* - \hat{g}_\star\|^2 \le \|u^\dagger - \hat{g}_\star\|^2$. Expanding both sides via $\hat{g}_\star = g_\star + \xi$:

$$\|u^* - g_\star\|^2 - 2\langle u^* - g_\star, \xi\rangle \le \|u^\dagger - g_\star\|^2 - 2\langle u^\dagger - g_\star, \xi\rangle.$$

Rearranging, we get

$$\|u^* - g_\star\|^2 \le \|u^\dagger - g_\star\|^2 + 2\langle u^* - u^\dagger, \xi\rangle \le \|u^\dagger - g_\star\|^2 + 4 \sup_{u \in U} |\langle \xi, u\rangle|.$$

Taking expectation over randomness in U (i.e., over B_t), we have

$$\mathrm{MSE}(u_t^{\mathrm{glob}}) \le \mathbb{E}[\|u^\dagger - g_\star\|^2 \mid \theta_t] + 4\mathbb{E}\left[\sup_{u \in U} |\langle \xi, u\rangle| \middle| \theta_t\right],$$

The first term is simply $\mathcal{B}(U) \coloneqq \mathbb{E}[\inf_{u \in U} \|u - g_\star\|^2 \mid \theta_t]$. Hence, we have

$$\mathcal{V}(u_t^{\mathrm{glob}}) = \mathrm{MSE}(u_t^{\mathrm{glob}}) - \mathcal{B}(U) \le 4\mathbb{E}\left[\sup_{u \in U} |\langle \xi, u\rangle| \middle| \theta_t\right],$$

and it remains to bound $\mathbb{E}[\sup_{u \in U} |\langle \xi, u\rangle| \mid \theta_t]$. By the sub-Gaussian assumption on the target noise, for each fixed u the random variable $\langle \xi, u\rangle$ is sub-Gaussian with parameter $\sigma^2 \|u\|^2/m$, i.e.,

$$\mathbb{E}\left[\exp(\lambda \langle \xi, u\rangle) \middle| \theta_t\right] \le \exp\left(\frac{\lambda^2 \sigma^2}{2m} \|u\|^2\right), \quad \forall \lambda \in \mathbb{R}.$$

By Assumption 2 and convexity, $\|u\| \le G$ for all $u \in U$. The sub-Gaussian maximal inequality [Vershynin, 2018] states that for N mean-zero sub-Gaussian random variables $X_1, \ldots, X_N$ with common parameter τ^2, $\mathbb{E}[\max_i |X_i|] \le \tau\sqrt{2\log(2N)}$. Applying this with $N = |U| \le \binom{n}{k}$ and $\tau^2 = \sigma^2 G^2/m$:

$$\mathbb{E}\left[\sup_{u \in U} |\langle \xi, u\rangle| \middle| \theta_t\right] \le \frac{\sigma G}{\sqrt{m}} \sqrt{2\log(2|U|)}.$$

Taking expectation over ξ (i.e., over $B_t^\star$) finishes the proof.

(ii) *Part (ii) (Group-Wise Subset Update).* The argument follows Part (i). Write $U \coloneqq U_t^{\mathrm{grp}}(k)$. By Assumption 2, each per-group component satisfies $\|u^{(p)}\| \le G$, so $\|u\|^2 \le PG^2$. Hence, the same argument yields

$$\mathcal{V}(u_t^{\mathrm{grp}}) \le \frac{4G\sqrt{P}\sigma}{\sqrt{m}} \sqrt{2\log(2|U|)}.$$

The cardinality bound follows from independence across groups: each group p independently chooses $S_{t,p} \subset [n]$ with $|S_{t,p}| = k$, so $|U| \le \binom{n}{k}^P$. Substituting: $\log(2\binom{n}{k}^P) = \log 2 + P\log\binom{n}{k} \le P\log(2\binom{n}{k})$ for $P \ge 1$, so $G\sqrt{P} \cdot \sqrt{2P\log(2\binom{n}{k})} = GP\sqrt{2\log(2\binom{n}{k})}$.

□

Simulation. Overall, which update dominates depends on the target batch size, the training–target mismatch, and the feasible-set complexity. Figure 12 illustrates three representative regimes using synthetic parameter choices to illustrate Theorem 3.4:

- **Small mismatch.** All four methods occupy distinct regimes. As m grows, the preferred update transitions from the Full-Training Update to the Global Subset Update, then to the Group-Wise Subset Update, and finally to the Target-Only Update.

- **Moderate mismatch.** The Group-Wise Subset Update dominates the Global Subset Update across a wider range of m, yielding a transition from the Full-Training Update to the Group-Wise Subset Update and then to the Target-Only Update.
- **Large mismatch.** The bias of full-training and Global Subset Updates is large enough that the Group-Wise Subset Update dominates even for relatively small m, before eventually giving way to the Target-Only Update when enough target data are available.

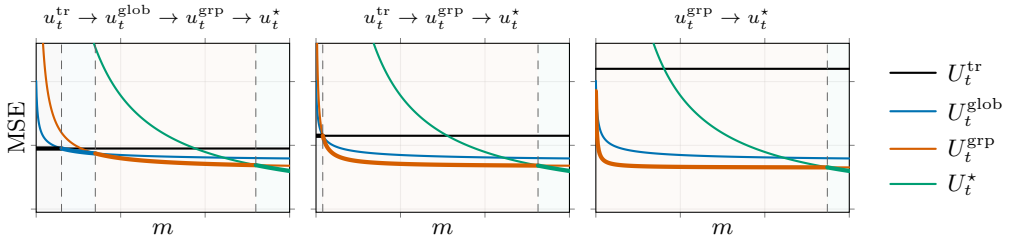

Figure 12: Illustration of MSE for each method as a function of target sample size m, according to Theorem 3.4. Shaded regions indicate the method with the smallest MSE. As distribution mismatch increases (left to right), Group-Wise Subset Update dominates over a wider range of m.

C Omitted Details from Section 4

This section provides the details deferred from Section 4. First, Section C.1 fills the gap of Section 4.2, where we discuss the tensor lifetime scheduling and also the scoring for general partition that we omit in Section 4. Section C.2 then presents a naive implementation that avoids memory management issues by using two passes. Section C.3 explains how to compress per-sample gradients efficiently without materializing the full gradient. Section C.4 derives the exact computational and memory complexity for each scoring method in Table 1. Section C.5 then describes how Layer-Wise Subset Update integrates with LoRA and MeSO, covering gradient representations and optimizer state transfer. Finally, Section C.6 discusses how data regularization extends beyond linear layers, including an efficient scoring algorithm for embedding layers and the treatment of other non-linear modules, specifically normalization layers.

Notation. Unlike in the main text where we focus primarily on layer-aligned partition $\mathcal{G} = \{G_p\}_{p=1}^P$ such that G_p contains parameters of layer(s), in this section, we consider general partitions such that each group might contain a part of each layer and can also span across multiple layers. For the ease of presentation, we overload the notation $^{(p)}$ and $^{(l)}$, where the former always refers to group-wise quantities, and the latter refers to layer-wise quantities.

C.1 General Partition

The one-pass tensor lifetime schedule presented in Section 4.2 focuses on layer-aligned partitions, where each group G_p is a union of whole layers. Here, we discuss the additional complications that arise for arbitrary partitions whose groups may split parameters within a single layer.

C.1.1 Layer-by-Layer Discipline

After the backward pass, all retained pairs $(a^{(l)}, \partial\ell/\partial e^{(l)})$ are available. To stay within the memory budget, score computation and gradient assembly must still proceed layer by layer. At each layer l, we compute per-layer score contributions and accumulate them into the scores for each group. A group G_p becomes resolvable once all layers intersecting G_p have been visited, at which point $S_{t,p}$ is determined. The update direction $u^{(p)}$ is then assembled from the per-sample gradients of the selected samples at layers intersecting G_p, and the retained quantities at layer l are released once all groups intersecting that layer have been resolved.

C.1.2 Non-Layer-Aligned Partitions

The layer-by-layer discipline introduces a tension between memory and recomputation for partitions that do not align with the layer structure. When a group G_p spans multiple layers, $S_{t,p}$ is not known until all layers in the group have been scored. If the scoring strategy materializes per-sample gradients $g_i^{(l)}$ at each layer, these cannot in general be retained across the group's full span without exceeding the memory budget; they must be discarded after scoring, and recomputed from the retained pairs $(a^{(l)}, \partial\ell/\partial e^{(l)})$ when $u_t^{(p)}$ is assembled.

Scoring. We now show how to compute the per-group scores $s_i^{(p)}$ during the backward pass. A crucial observation is that the score decomposes linearly over *parameter space*, i.e., $s_i^{(p)} = \sum_{q \in G_p} g_{i,q} \cdot \hat{g}_{\star,q}$ where $g_{i,q}$ denotes the q^{th} entry of g_i, similarly for $\hat{g}_{\star,q}$. Hence, computing $s_i^{(p)}$ reduces to computing the *per-parameter* score $s_{i,q} := g_{i,q} \cdot \hat{g}_{\star,q}$ for each $i \in [n]$ and $q \in [d]$. This is realized via layer-wise accumulation: for each p, we initialize $s_i^{(p)} \leftarrow 0$. Then during the backward pass at layer l, we first materialize the per-sample gradient $g_i^{(l)}$ and the average target gradient $\hat{g}_\star^{(l)}$, and obtain all per-parameter scores $s_{i,q}$ by computing $\hat{g}_\star^{(l)} \odot g_i^{(l)} \in \mathbb{R}^{d/L}$ for all q belonging to layer l. Finally, we accumulate $s_i^{(p)} \leftarrow s_i^{(p)} + \sum_{q \in G_p} s_{i,q}$, and continue the backward pass to layer $l+1$.

Remark C.1. *We highlight that the scoring methods discussed in Section 4.3, except for Direct, therefore can not handle general partition, as they do not materialize $g_\star^{(l)}$ and $g_i^{(l)}$ explicitly.*

Update assembly. Recall that for layer-aligned partitions, we can break down the construction of $u_t^{(p)}$ into layers. Similarly, in the case of general partitions, we again follow the layer-by-layer discipline, where for each layer, we first identify groups G_p that have non-empty intersections with it, and for each of these groups, what are the corresponding parameters in the intersection. Then, we materialize all the per-sample gradients $\{g_i^{(l)}\}_{i=1}^n$ of the entire layer, and resolve for each group by first computing the average over samples $i \in S_{t,p}$ with sub-vectors of $g_i^{(l)}$ that corresponds to parameters that belongs to the group.

C.2 Two Pass Implementation

Recall from Section 4.1 that in standard training, both $a^{(l)}$ and $\partial\ell/\partial e^{(l)}$ can be released as soon as the batch gradient $\hat{g}_{\text{tr}}^{(l)}$ is assembled, that is, once the update direction at layer l is decided. For Group-Wise Subset Update, however, the update direction at each group p depends on $S_{t,p}$, and $S_{t,p}$ can only be determined after scoring all samples across every layer whose parameters intersect G_p. Standard training releases the per-sample quantities needed for gradient assembly before $S_{t,p}$ is known. For Layer-Wise Subset Update, each group corresponds to a single layer, so $S_{t,p}$ can be determined after scoring all samples within that layer, and per-sample quantities can be released immediately. For coarser partitions, however, a two-pass approach trades extra computation for standard memory.

C.2.1 Separate Scoring and Gradient Passes

One solution is to use two passes. A first *scoring pass* accumulates per-group scores $s_i^{(p)}$ layer by layer during a standard forward-backward pass as described in Section C.1. After $S_{t,p}$ is determined, a second *gradient pass* re-runs forward-backward on the union $\bigcup_p S_{t,p}$: at each layer l, the gradient for each group p whose parameters intersect that layer is assembled from the samples in $S_{t,p}$, which again follow the same strategy described in Section C.1 in the case of general partitions.

As a concrete instance, consider Global Subset Update, where all parameters form a single group so that $S_{t,p} = S_t$ is global. The per-sample score decomposes as $s_i = \sum_{l=1}^L s_i^{(l)}$, so the scoring pass accumulates per-layer contributions into one global score per sample. The gradient pass simplifies to a forward-backward on the k selected samples, at an additional cost of recomputation. GREATS [Wang et al., 2024b] adopts this two-pass design; Algorithm C.1 presents the full pseudocode.

Algorithm C.1: Global Subset Update (two-pass)

Data: Model θ_t, training data $\{z_i\}_{i=1}^n$, target data $\{z_j^\star\}_{j=1}^m$, learning rate η_t
Result: Updated model parameters θ_{t+1}

/* Scoring pass */
$(\ell, \{a_i^{(l)}, e_i^{(l)}\}_{l,i} \cup \{a_j^{\star(l)}, e_j^{\star(l)}\}_{l,j}) \leftarrow$ `Forward`$(\theta_t, \{z_i\}_{i=1}^n \cup \{z_j^\star\}_{j=1}^m)$ // $+O(2NT\sqrt{dL})$
for $l = L, \ldots, 1$ **do**
  $\{\partial \ell / \partial e_i^{(l)}\}_{i=1}^n \cup \{\partial \ell / \partial e_j^{\star(l)}\}_{j=1}^m \leftarrow$ `Backward`$(\ell, \theta_t^{(l)})$
  $s_i^{(l)} \leftarrow$ `Score`$(a_i^{(l)}, \partial \ell / \partial e_i^{(l)}, \{a_j^{\star(l)}, \partial \ell / \partial e_j^{\star(l)}\}_{j=1}^m)$ for each $i \in [n]$ // Section 4.3
  `Release`$(a_i^{(l)}, \partial \ell / \partial e_i^{(l)}, a_j^{\star(l)}, \partial \ell / \partial e_j^{\star(l)})$ for all i, j // $-O(2NT\sqrt{d/L})$
$s_i \leftarrow \sum_{l=1}^L s_i^{(l)}$ for all $i \in [n]$ // Global scores
$S_t \leftarrow$ `Select-S`$(\{s_i\}_{i=1}^n, k)$
/* Gradient pass on subset S_t */
$(\ell_{S_t}, \{a_i^{(l)}\}_{l,i}) \leftarrow$ `Forward`$(\theta, \{z_i\}_{i \in S_t})$ // $+O(kT\sqrt{dL})$
for $l = L, \ldots, 1$ **do**
  $\{\partial \ell_{S_t} / \partial e_i^{(l)}\}_{i \in S_t} \leftarrow$ `Backward`$(\ell_{S_t}, \theta_t^{(l)})$ // $-O(kT\sqrt{d/L}) + O(kT\sqrt{d/L})$
  $u_t^{(l)} \leftarrow \frac{1}{k} \sum_{i \in S_t} (\partial \ell_{S_t} / \partial e_i^{(l)}) \otimes a_i^{(l)}$ // $+O(d/L)$
  `Release`$(a_i^{(l)}, \partial \ell_{S_t} / \partial e_i^{(l)})$ for all $i \in S_t$ // $-O(2kT\sqrt{d/L})$
  $\theta_{t+1}^{(l)} \leftarrow \theta_t^{(l)} - \eta_t u_t^{(l)}$
  `Release`$(u_t^{(l)})$ // $-O(d/L)$
return θ_{t+1}

C.2.2 Compatibility with Memory-Saving Techniques

The two-pass design is compatible with both activation checkpointing and gradient accumulation, since each pass independently preserves the standard memory profile. For activation checkpointing, when the partition does not align with the checkpointing schedule (i.e., some group G_p spans multiple checkpoint segments), the scoring pass computes per-group scores under standard checkpointing, releasing activations at segment boundaries, and the gradient pass performs a checkpointed forward-backward over $\bigcup_p S_{t,p}$. For gradient accumulation with rules that depend on global batch statistics (e.g., top-k), the scoring pass processes all micro-batches to determine $S_{t,p}$ for every group, and the gradient pass assembles gradients from $\bigcup_p S_{t,p}$, itself split into micro-batches if needed.

C.3 Efficient Per-Sample Gradient Projection

In this section, we provide a self-contained discussion on the state-of-the-art per-sample gradient compression [Hu et al., 2025a], which is used in both the MeSO integration and also the compressed scoring, where the scoring is computed using compressed gradients.

C.3.1 Factorized Projection

State-of-the-art gradient compressors [Hu et al., 2025a, Choe et al., 2025] consider a projection operator $\Pi \in \mathbb{R}^{\kappa \times d}$ from dimension $d := d_{\text{in}} d_{\text{out}}$ to dimension $\kappa \ll d$ with a two-stage structure:

$$\Pi = P_{\text{final}} \left(P_{\text{in}} \otimes P_{\text{out}} \right),$$

where $P_{\text{in}} \in \mathbb{R}^{\kappa_{\text{in}} \times d_{\text{in}}}$ and $P_{\text{out}} \in \mathbb{R}^{\kappa_{\text{out}} \times d_{\text{out}}}$ are the first-stage projections, $P_{\text{in}} \otimes P_{\text{out}}$ denotes their Kronecker product, and $P_{\text{final}} \in \mathbb{R}^{\kappa \times \kappa_{\text{in}} \kappa_{\text{out}}}$ is a small second-stage projection.

Remarkably, for any input vectors that admit such a factorized structure, the factorized projection can be computed efficiently without materializing the full $\mathbb{R}^{\kappa \times d}$ representation of Π, and can directly operate on the input factors in $\mathbb{R}^{d_{\text{in}}}$ and $\mathbb{R}^{d_{\text{out}}}$, enabling efficient forward and back-projection. This allows us to apply such a compressor to linear-layer gradients efficiently, as we will soon see.

Forward projection. For any input with outer-product structure $b \otimes a$ where $a \in \mathbb{R}^{d_{\text{in}}}$ and $b \in \mathbb{R}^{d_{\text{out}}}$, the forward projection evaluates as

$$\Pi \operatorname{vec}(b \otimes a) = P_{\text{final}} \operatorname{vec}\big((P_{\text{out}} b) \otimes (P_{\text{in}} a)\big),$$

requiring only two small matrix-vector products ($P_{\text{in}} a \in \mathbb{R}^{\kappa_{\text{in}}}$ and $P_{\text{out}} b \in \mathbb{R}^{\kappa_{\text{out}}}$) followed by the second-stage projection, without forming $\operatorname{vec}(b \otimes a) \in \mathbb{R}^d$. By linearity, this extends to sums of outer products: for $X = \sum_\tau b_\tau \otimes a_\tau$,

$$\widetilde{X} = \Pi \operatorname{vec}(X) = P_{\text{final}} \operatorname{vec}\left(\sum_\tau (P_{\text{out}} b_\tau) \otimes (P_{\text{in}} a_\tau)\right),$$

where each term contributes a small outer product in $\mathbb{R}^{\kappa_{\text{out}} \times \kappa_{\text{in}}}$.

Backward projection. For any vector $\widetilde{x} \in \mathbb{R}^\kappa$, the transpose map $\Pi^\top \widetilde{x} \in \mathbb{R}^d$ is equally efficient. Let $X' \in \mathbb{R}^{\kappa_{\text{out}} \times \kappa_{\text{in}}}$ be the matrix obtained by reshaping $(P_{\text{final}})^\top \widetilde{x}$. The Kronecker structure gives

$$\operatorname{Mat}(\Pi^\top \widetilde{x}) = (P_{\text{out}})^\top X' P_{\text{in}},$$

a pair of small matrix multiplications, without forming the full matrix $\Pi^\top \in \mathbb{R}^{d \times \kappa}$.

C.3.2 Per-Sample Gradient Compression

We now show how to apply this compressor to compress linear layers' per-sample gradients.

Notation. For generality, we do not assume that each linear layer is square: specifically, the weight at layer l is $W^{(l)} \in \mathbb{R}^{d_{\text{out}}^{(l)} \times d_{\text{in}}^{(l)}}$ with $d^{(l)} \coloneqq d_{\text{in}}^{(l)} d_{\text{out}}^{(l)}$, related to the vectorized form by $\theta^{(l)} = \operatorname{vec}(W^{(l)})$. Per-training-sample weight gradients are correspondingly denoted $G_i^{(l)} \in \mathbb{R}^{d_{\text{out}}^{(l)} \times d_{\text{in}}^{(l)}}$, so that $g_i^{(l)} = \operatorname{vec}(G_i^{(l)})$. Similarly for $G_j^\star$ for per-target-sample gradients. Let $\langle \cdot, \cdot \rangle$ denote the Frobenius inner product, i.e., the Euclidean inner product of the vectorized forms.

Efficient per-sample gradient compression. Recall that the per-sample weight gradient for sample z_i at layer l is a sum of outer products over tokens: $g_i^{(l)} = \sum_{\tau=1}^T (\partial \ell / \partial e_{i,\tau}^{(l)}) \otimes a_{i,\tau}^{(l)} \in \mathbb{R}^{d^{(l)}}$. Consider a compressor $\Pi^{(l)} \coloneqq P_{\text{final}}^{(l)}(P_{\text{in}}^{(l)} \otimes P_{\text{out}}^{(l)})$, with $P_{\text{in}}^{(l)} \in \mathbb{R}^{\kappa_{\text{in}}^{(l)} \times d_{\text{in}}^{(l)}}$ and $P_{\text{out}}^{(l)} \in \mathbb{R}^{\kappa_{\text{out}}^{(l)} \times d_{\text{out}}^{(l)}}$, and $P_{\text{final}}^{(l)} \in \mathbb{R}^{\kappa^{(l)} \times \kappa_{\text{in}}^{(l)} \kappa_{\text{out}}^{(l)}}$. Applying the forward projection gives the compressed per-sample gradient

$$\widetilde{g}_i^{(l)} = P_{\text{final}}^{(l)} \operatorname{vec}\left(\sum_{\tau=1}^T \left(P_{\text{in}}^{(l)} a_{i,\tau}^{(l)}\right) \otimes \left(P_{\text{out}}^{(l)} \frac{\partial \ell}{\partial e_{i,\tau}^{(l)}}\right)\right) \in \mathbb{R}^{\kappa^{(l)}}.$$

We note that this can be computed directly from the cached activations and activation gradients, without materializing $g_i^{(l)}$. With careful choice of $P_{\text{in}}^{(l)}$, $P_{\text{out}}^{(l)}$, and $P_{\text{final}}^{(l)}$, the per-sample memory is $O(\kappa^{(l)})$ instead of $O(d^{(l)})$, and the per-token computational cost is $O(\kappa^{(l)})$ [Hu et al., 2025a], so compressing all n samples across T tokens and L layers costs $O(nT\kappa)$ in total where $\kappa = \sum_l \kappa^{(l)}$. Later, we write this operation as $\Pi^{(l)}(a_i^{(l)}, \partial \ell / \partial e_i^{(l)}, \{a_j^{\star(l)}, \partial \ell / \partial e_j^{\star(l)}\})$ in Algorithm C.2 to emphasize that $\Pi^{(l)}$ operates on factorized inputs without materializing the full gradient.

C.4 Scoring Complexity

We derive the exact computational and memory complexity for each scoring method in Table 1. Every addition and multiplication counts as one operation; memory is measured in scalar entries under fully batched execution. Throughout, we let $w = \sqrt{d/L}$ to denote the layer width for brevity, $N = n + m$, and all quantities carry a layer index $^{(l)}$ that we suppress for brevity.

C.4.1 Direct

Three steps:

1. *Materialize n training gradient matrices* $g_i = \sum_{\tau=1}^{T} (\partial \ell / \partial e_{i,\tau}) \otimes a_{i,\tau}$. Each $\otimes$ Kronecker product costs w^2 multiplications; summing T outer products element-wise adds $(T-1)w^2$ additions. Per sample: $(2T-1)w^2$. Total: $n(2T-1)w^2$.
2. *Materialize* $\hat{g}_\star = \frac{1}{m} \sum_j g_j^\star$. Same Kronecker-product accumulation for m target samples: $m(2T-1)w^2$. Summing the m matrices: $(m-1)w^2$ additions. Scaling by $1/m$: w^2 multiplications. Total: $m(2T-1)w^2 + (m-1)w^2 + w^2 = 2mTw^2$.
3. *Score.* Each Frobenius inner product $s_i = \langle \hat{g}_\star, g_i \rangle$ costs w^2 multiplications + (w^2-1) additions $= (2w^2-1)$ per sample. Total: $n(2w^2-1)$.

FLOPs. $n(2T-1)w^2 + 2mTw^2 + n(2w^2-1) = 2NTw^2 + n(w^2-1)$.

Memory. n gradient matrices $g_i \in \mathbb{R}^{w \times w}$ plus $\hat{g}_\star$: $(n+1)w^2$.

C.4.2 Ghost Inner Product (GIP)

For each training-target pair (i,j):

1. *Two $T \times T$ matrix products.* Form the activation-gradient cross-correlation $(\partial \ell / \partial e_i)^\top (\partial \ell / \partial e_j^\star) \in \mathbb{R}^{T \times T}$ and the activation cross-correlation $(a_i)^\top a_j^\star \in \mathbb{R}^{T \times T}$. Each entry is a dot product of w-vectors: w multiplications + $(w-1)$ additions $= (2w-1)$ per entry, $T^2(2w-1)$ per matrix. Two such matrices: $2T^2(2w-1)$.
2. *Frobenius inner product.* Element-wise multiply and sum: T^2 multiplications + (T^2-1) additions $= (2T^2-1)$.

Per pair: $2T^2(2w-1) + (2T^2-1) = 4T^2w - 1$. Accumulating m pairs per training sample adds $(m-1)$ additions and 1 multiplication (the $1/m$ scaling), giving $m(4T^2w-1) + m = 4mT^2w$ per training sample.

FLOPs. $4nmT^2w$.

Memory. Two $T \times T$ cross-correlation matrices per (i,j) pair, batched over all pairs: $2nmT^2$.

C.4.3 Per-Token Inner Product (PIP)

Two steps:

1. *Materialize* $\hat{G}_\star$. Identical to Step 2 of direct: $2mTw^2$.
2. *Score.* For each training token (i,τ): a matrix-vector product $\hat{G}_\star a_{i,\tau} \in \mathbb{R}^w$ costs $w(2w-1)$ (each of w entries is a dot product of w-vectors), then a dot product with $\partial \ell / \partial e_{i,\tau}$ costs $(2w-1)$. Per token: $w(2w-1) + (2w-1) = (w+1)(2w-1) = 2w^2 + w - 1$. Accumulating T tokens into s_i: $T(2w^2+w-1) + (T-1) = 2Tw^2 + Tw - 1$ per sample. Total: $n(2Tw^2 + Tw - 1)$.

FLOPs. $2mTw^2 + n(2Tw^2 + Tw - 1) = 2NTw^2 + n(Tw-1)$.

Memory. $\hat{G}_\star \in \mathbb{R}^{w \times w}$ plus the batched intermediates $\hat{G}_\star A \in \mathbb{R}^{w \times nT}$: $w^2 + nTw$.

C.4.4 Compressed Gradient

Three steps:

1. *Compress all N per-sample gradients* to $\widetilde{g}_i \in \mathbb{R}^{\kappa^{(l)}}$ via the projection Π. The per-token cost depends on the Kronecker structure of Π (see Section C.3); the total is $O(NT\kappa^{(l)})$.
2. *Aggregate target gradient* $\widetilde{g}_\star = \frac{1}{m} \sum_j \widetilde{g}_j^\star$. Summing m vectors of dimension $\kappa^{(l)}$: $(m-1)\kappa^{(l)}$ additions. Scaling by $1/m$: $\kappa^{(l)}$ multiplications. Total: $m\kappa^{(l)}$.
3. *Score.* Each inner product $s_i = \langle \widetilde{g}_\star, \widetilde{g}_i \rangle$ costs $\kappa^{(l)}$ multiplications + $(\kappa^{(l)}-1)$ additions $= (2\kappa^{(l)}-1)$ per sample. Total: $n(2\kappa^{(l)}-1)$.

FLOPs. $O(NT\kappa^{(l)}) + (2n+m)\kappa^{(l)}$.

Memory. n compressed training gradients plus $\widetilde{g}_\star$: $(n+1)\kappa^{(l)}$.

C.5 Integrating LoRA and MeSO

The Dr. Post-Training framework is agnostic to the gradient representation at each layer, composing naturally with parameter-efficient and memory-efficient training methods. We now detail the integration with LoRA and MeSO.

C.5.1 LoRA Integration

We adapt the same notation as in Section A.1. Since the two LoRA adapters $B^{(l)}$ and $A^{(l)}$ are themselves linear maps, the per-sample gradients admit the same outer-product factorization as full-parameter gradients:

$$G_i^{(A,l)} = \sum_{\tau=1}^{T} \frac{\partial \ell}{\partial a_{i,\tau}^{(B,l)}} \otimes a_{i,\tau}^{(B,l)} \in \mathbb{R}^{r \times d_{\text{in}}^{(l)}}, \qquad G_i^{(B,l)} = \sum_{\tau=1}^{T} \frac{\partial \ell}{\partial e_{i,\tau}^{(l)}} \otimes a_{i,\tau}^{(A,l)} \in \mathbb{R}^{d_{\text{out}}^{(l)} \times r}.$$

The layer-wise scoring $s_i^{(l)} = \langle \hat{G}_\star^{(l)}, G_i^{(l)} \rangle$ and gradient computation proceed identically to the full-parameter case, but with the adapter dimension $r(d_{\text{in}}^{(l)} + d_{\text{out}}^{(l)})$ replacing the full dimension $d_{\text{in}}^{(l)} d_{\text{out}}^{(l)}$. This yields proportional reductions in both scoring and gradient computation cost. No modification to the Layer-Wise Subset Update algorithm (Algorithm 4.3) is needed; LoRA simply changes the gradient representation that the algorithm operates on.

C.5.2 MeSO Integration

We adapt the same notation as in Section A.2. We see that since MeSO only changes how the optimizer update is applied, not how per-sample gradients are computed, data regularization integrates directly: any scoring method from Section 4.3 (including the exact methods) can be used to determine $S_{t,p}$, and the optimizer then updates in the projected subspace using only the samples in $S_{t,p}$.

Compression. We adapt the Section C.3 for the computation of the compression required by MeSO. Note that the compressed gradients are trivial to compute by directly applying the forward projection in the factorized form. The tricky part is that when computing the momentum states transfer during subspace refreshing, some of the input does not have the factorized structure.

Taking the first moment $\hat{m}_{\text{old}}$ as an example, we see that the transformation requires computing $\Pi_{\text{new}}\Pi_{\text{old}}^\top \hat{m}_{\text{old}}$. We see that for the backward projection, we have an efficient implementation to compute $\Pi_{\text{old}}^\top \hat{m}_{\text{old}}$. However, for the forward projection, it is unclear how to forward project this general input $\Pi_{\text{old}}^\top \hat{m}_{\text{old}}$ under Π_{new}. However, a simple application of the same trick (formally known as the *vec-trick*), as discussed in the backward projection, resolves this problem and provides a way to compute the forward projection on general input. We omit the details here for simplicity.

Scoring. For exact scoring, this is quite straightforward: we first compute the scores using one of the exact scoring methods discussed in Section 4.3, discard all the intermediate quantities, solve for $S_{t,p}$, and finally compute the compressed gradients for $S_{t,p}$. Since the exact scoring and compressed gradient computations generally do not have overlapping computations, we do not need to consider retaining any intermediate quantities to avoid recomputation, simplifying the memory management issue.

On the other hand, for approximate scoring, we see that the compressed per-sample gradients that MeSO computes for parameter updates can also be used for compressed scoring (Section 4.3.3) as an additional optimization, so that the *same* compressed gradients $\widetilde{g}_i^{(l)}, \widetilde{g}_\star^{(l)}$ drive both alignment scoring ($s_i^{(l)} = \langle \widetilde{g}_\star^{(l)}, \widetilde{g}_i^{(l)} \rangle$) and the optimizer update, avoiding redundant computation.

However, in this case, we note that managing memory for the compressed per-sample gradients needed by the optimizer update and compressed scoring is again non-trivial: when there is a group G_p that contains many layers, materializing all the per-sample compressed gradients across all layers may still be memory-intensive and infeasible under a tight memory budget.

Memory management for compressed scoring. We next discuss the memory-management aspect for MeSO with compressed gradients reused. Specifically, assume that we materialize all the $O(n)$ compressed per-sample gradients $\widetilde{g}_i^{(l)}$ and also $\widetilde{g}_\star^{(l)}$ during the backward pass. Once $\widetilde{g}_i^{(l)}$ is computed, the activations $a_i^{(l)}$ and activation gradients $\partial\ell/\partial e_i^{(l)}$ can be released immediately, since the optimizer update only requires averaging the compressed gradients in $S_{t,l}$. This replaces $O(2nT\sqrt{d/L})$ memory (for $a_i^{(l)}$ and $\partial\ell/\partial e_i^{(l)}$) with $O(n\kappa^{(l)})$ (for $\widetilde{g}_i^{(l)}$), a net savings when $\kappa^{(l)} < T\sqrt{d/L}$, which holds in typical settings. Hence, it is generally possible to directly retain all the compressed gradients to avoid recomputation of the per-sample compressed gradients for the MeSO updates.

For illustration, the pseudocode for this strategy in the case of Layer-Wise Subset Update is provided in Algorithm C.2, where the compressed gradients are further released at each layer once $S_{t,l}$ is determined, so only one layer's worth of per-sample compressed gradients is held at a time.

Algorithm C.2: Layer-Wise Subset Update with MeSO

Data: Model θ_t, training data $\{z_i\}_{i=1}^n$, target data $\{z_j^\star\}_{j=1}^m$, learning rate η_t, projections Π
Result: Updated model parameters θ_{t+1}

$(\ell, \{a_i^{(l)}\}_{l,i}, \{a_j^{\star(l)}\}_{l,j}) \leftarrow$ `Forward(`θ_t, $\{z_i\}_{i=1}^n \cup \{z_j^\star\}_{j=1}^m$`)`
for $l = L, \dots, 1$ **do**
  $\{\partial\ell/\partial e_i^{(l)}\}_{i=1}^n, \{\partial\ell/\partial e_j^{\star(l)}\}_{j=1}^m \leftarrow$ `Backward(`ℓ, $\theta_t^{(l)}$`)`
  $\widetilde{g}_i^{(l)} \leftarrow \Pi(a_i^{(l)}, \partial\ell/\partial e_i^{(l)}) \in \mathbb{R}^{\kappa^{(l)}}$ for all $i \in [n]$ `// Compress training`
  $\widetilde{g}_j^{\star(l)} \leftarrow \Pi(a_j^{\star(l)}, \partial\ell/\partial e_j^{\star(l)}) \in \mathbb{R}^{\kappa^{(l)}}$ for all $j \in [m]$ `// Compress target`
  `Release(`$a_i^{(l)}$, $\partial\ell/\partial e_i^{(l)}$, $a_j^{\star(l)}$, $\partial\ell/\partial e_j^{\star(l)}$`)` for all i, j
  $\widetilde{g}_\star^{(l)} \leftarrow \frac{1}{m}\sum_{j=1}^m \widetilde{g}_j^{\star(l)}$
  $s_i^{(l)} \leftarrow \langle \widetilde{g}_\star^{(l)}, \widetilde{g}_i^{(l)} \rangle$ for each $i \in [n]$ `// Score via compressed gradients`
  $S_{t,l} \leftarrow$ `Select-S(`$\{s_i^{(l)}\}_{i=1}^n$, k`)`
  $\widetilde{u}_t^{(l)} \leftarrow \frac{1}{k}\sum_{i \in S_{t,l}} \widetilde{g}_i^{(l)}$ `// Aggregate selected`
  `Release(`$\widetilde{g}_i^{(l)}$`)` for all i
  $\theta_{t+1}^{(l)} \leftarrow \theta_t^{(l)} - \eta_t \cdot (\Pi^\top \widetilde{u}_t^{(l)})$ `// Back-project and update`
  `Release(`$\widetilde{u}_t^{(l)}$`)`
return θ_{t+1}

C.6 Beyond Linear Layers

The system design in Section 4 develops scoring and selection exclusively for linear layers (e.g., `nn.Linear` in `PyTorch`), which account for the vast majority of trainable parameters in standard transformer architectures. A transformer model, however, also contains other trainable modules—token embeddings (`nn.Embedding`) and normalization layers (`nn.LayerNorm`, `nn.RMSNorm`)—whose treatment we now discuss.

C.6.1 Embedding Layers

The token embedding layer maps each input token index $x_{i,\tau} \in [V]$ to a dense vector via a lookup: $\mathrm{output}_{i,\tau} = W[x_{i,\tau}]$, where $W \in \mathbb{R}^{V \times D}$ is the embedding matrix, V is the vocabulary size, and D is the embedding dimension. Unlike a linear layer, whose per-sample gradient is a sum of dense outer products, the embedding gradient is *sparse*: writing $\delta_{i,\tau} := \partial\ell_i/\partial\mathrm{output}_{i,\tau} \in \mathbb{R}^D$ for the activation gradient at position τ, the per-sample weight gradient is

$$G_i = \sum_{\tau=1}^{T} e_{x_{i,\tau}} \otimes \delta_{i,\tau} \in \mathbb{R}^{V \times D},$$

where $e_{x_{i,\tau}} \in \mathbb{R}^V$ is the one-hot indicator for token $x_{i,\tau}$. Row v of G_i equals $\sum_{\tau:\, x_{i,\tau}=v} \delta_{i,\tau}$, and all other rows are zero.

Efficient scoring via lookup. This sparsity enables a scoring algorithm that avoids materializing any per-sample gradient matrix. The target embedding gradient is

$$\hat{G}_\star = \frac{1}{m}\sum_{j=1}^{m}\sum_{\tau=1}^{T} e_{x^\star_{j,\tau}} \otimes \delta^\star_{j,\tau} \in \mathbb{R}^{V\times D},$$

computable in $O(mTD)$ via `index_add` (scatter-accumulate over token indices). The per-sample alignment score then simplifies as follows:

$$s_i^{(\mathrm{emb})} = \langle G_i, \hat{G}_\star\rangle = \sum_{v=1}^{V}\langle G_i[v], \hat{G}_\star[v]\rangle = \sum_{\tau=1}^{T}\langle \delta_{i,\tau}, \hat{G}_\star[x_{i,\tau}]\rangle,$$

where the last equality follows from $G_i[v] \neq 0$ only for $v \in \{x_{i,\tau}\}_\tau$. Each term is a table lookup of $\hat{G}_\star$ at index $x_{i,\tau}$ followed by a dot product with $\delta_{i,\tau}$, both $O(D)$ operations. Summing over T tokens gives $O(TD)$ per sample, and $O(nTD)$ across all n training samples. Including the cost of constructing $\hat{G}_\star$, the total is $O(NTD)$ FLOPs with $O(VD)$ memory for storing $\hat{G}_\star$.

Comparison with linear layer scoring. The embedding scoring algorithm is the natural analogue of the reduced ghost inner product for linear layers (Section 4.3). For a linear layer, the reduced ghost evaluates $s_i^{(l)} = \sum_\tau (\partial\ell/\partial e_{i,\tau}^{(l)})^\top \hat{G}_\star^{(l)} a_{i,\tau}^{(l)}$, which requires a matrix-vector product $\hat{G}_\star^{(l)} a_{i,\tau}^{(l)} \in \mathbb{R}^{\sqrt{d/L}}$ at each token. For the embedding layer, the "input" is a one-hot vector $e_{x_{i,\tau}}$, so the matrix-vector product $\hat{G}_\star\, e_{x_{i,\tau}} = \hat{G}_\star[x_{i,\tau}]$ reduces to a single row lookup, eliminating the $O(D)$ factor from the matrix-vector multiply. In practice, the activation gradients $\delta_{i,\tau}$ are already available during backpropagation (they are the upstream gradient flowing into the embedding layer), so scoring adds only the table-lookup and dot-product cost per token.

Gradient assembly. After selection, the weight gradient for the selected subset S is assembled as $G^{(\mathrm{emb})} = \frac{1}{|S|}\sum_{i\in S} G_i$, which is a sparse accumulation computable in $O(|S\|TD)$ via `index_add`, matching the cost of standard embedding gradient computation on a batch of size $|S\|$.

C.6.2 Normalization Layers

Normalization layers (e.g., `LayerNorm` [Ba et al., 2016], `RMSNorm` [Zhang and Sennrich, 2019]) contain a small number of trainable parameters—a scale vector $\gamma \in \mathbb{R}^D$ and optionally a bias $\beta \in \mathbb{R}^D$—contributing $O(D)$ parameters per normalization layer versus $O(D^2)$ for each linear layer. These parameters are not covered by the data regularization hooks and receive their gradients from the full batch via standard autograd, i.e., they are effectively trained under Full-Training Update ($U = \{\hat{g}_{\mathrm{tr}}\}$). In a typical transformer, the total parameter count of all normalization layers is negligible relative to that of the linear layers.

Summary. In our implementation, only `nn.Linear` modules within the transformer blocks are hooked for data-regularized training. The embedding layer admits efficient per-sample scoring via the lookup-based algorithm above, though it is not hooked in our current experiments (in LoRA and MeSO settings, the embedding is frozen; in the full-parameter setting, the `lm_head`, which is often weight-tied with the embedding, is included as a linear layer). Normalization layers are trained with the full batch. Since linear layers account for the vast majority of trainable parameters, this design covers the parameters that matter most while maintaining compatibility with standard training infrastructure.

D Omitted Details from Section 5

In this section, we provide all the omitted details for all experiment settings, including SFT, RLHF, and RLVR.

D.1 Computing Resources

Our experiments are conducted on the NCSA Delta cluster.[7] SFT, RLHF, and benchmark experiments run on nodes equipped with a single `AMD EPYC 7763 CPU @ 2.45GHz` (64 cores) and four `NVIDIA A40 GPUs` (48 GB GDDR6 each). RLVR experiments run on nodes with dual `Intel Xeon Platinum 8558 CPUs` (96 cores total) and eight `NVIDIA H200 GPUs` (141 GB HBM3 each).

D.2 Supervised Fine-Tuning

We now provide the missing details of the SFT experiments in Section 5.1, including details on datasets, training configurations, data regularization, and the case study.

D.2.1 Datasets

We consider four general/target pairs:

- `alpaca`[8] [Taori et al., 2023] (CC BY-NC 4.0 license)/`samsum`[9] [Gliwa et al., 2019] (CC BY-NC-ND 4.0 license). Single-turn instruction-following/dialogue summarization. We randomly sample 40% of the `alpaca` training pool per seed ($\sim$20.8k examples).
- `less-mix` [Xia et al., 2024a]/`tydiqa`[10] [Clark et al., 2020] (Apache-2.0 license). Heterogeneous instruction mixture/multilingual extractive QA. We randomly sample 0.5% of `less-mix` per seed ($\sim$9.8k examples).
- `triviaqa`[11] [Joshi et al., 2017]/`nq_open`[12] [Kwiatkowski et al., 2019] (CC BY-SA 3.0 license). Closed-book TriviaQA QA pairs (`rc.nocontext`, $\sim$138k examples)/closed-book Natural Questions evaluation. We randomly sample 5% of the `triviaqa` training pool per seed ($\sim$ 8.9k examples).
- `less-mix` [Xia et al., 2024a]/`squad`[13] [Rajpurkar et al., 2016] (CC BY-SA 4.0 license). Same training pool as the `tydiqa` setting, with the target task replaced by SQuAD reading-comprehension answers without context. 0.5% sampling rate as above.

For each target task we hold out 16 samples for data regularization, 500 samples for in-training perplexity tracking, and report the final-checkpoint downstream metric (Rouge-L for `samsum`; F1 for `tydiqa`, `nq_open`, and `squad`) on a separate 500-sample test split. `Llama-3.2-1B-Base` ships without a chat template, so we install an `open-instruct`-style fallback ($\langle$user$\rangle$/$\langle$assistant$\rangle$ plaintext markers) before tokenization, with the loss computed only on assistant-content tokens.

D.2.2 Training Configurations

All models are trained with standard supervised fine-tuning using next-token prediction on formatted instruction–response pairs. Across the four settings, we report results for (i) full-parameter fine-tuning, (ii) LoRA fine-tuning, and (iii) MeSO; `alpaca/samsum` reports all three fine-tuning methods, while `less-mix/tydiqa`, `triviaqa/nq_open`, and `less-mix/squad` report LoRA only. All methods use the same number of optimization steps within each dataset pair and the same effective batch size. We use AdamW [Loshchilov and Hutter, 2019] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, weight decay 0, a linear learning-rate schedule with 3% warmup, batch size 8, maximum sequence length 512, and train for 1 epoch. Training uses `bfloat16` precision and FlashAttention-2. For the learning rate, we follow Grattafiori et al. [2024], where we use 10^{-5} for full-parameter fine-tuning and MeSO, and 10^{-4} (10 times scaling [Schulman and Lab, 2025]).

The number of training steps is determined by the 1-epoch budget on the sampled training pool: $\sim$2600 for `alpaca/samsum`, $\sim$1225 for `less-mix/tydiqa` and `less-mix/squad`, and $\sim$1107 for `triviaqa/nq_open`. We log evaluation perplexity at $\sim$100 equally spaced steps across the trajectory in all four settings.

[7] https://docs.ncsa.illinois.edu/systems/delta/en/latest/index.html
[8] https://huggingface.co/datasets/tatsu-lab/alpaca
[9] https://huggingface.co/datasets/knkarthick/samsum
[10] https://huggingface.co/datasets/google-research-datasets/tydiqa
[11] https://huggingface.co/datasets/mandarjoshi/trivia_qa
[12] https://huggingface.co/datasets/google-research-datasets/nq_open
[13] https://huggingface.co/datasets/rajpurkar/squad

For LoRA, we use rank $r = 8$, scaling factor $\alpha = 16$, dropout 0.1, and apply adapters to all linear layers (the full Q/K/V/O attention projections and the gate/up/down MLP projections). For MeSO, we use the gradient compressor described in Section C.3.2 with a Gaussian projector for both $P_{\text{out}}^{(l)}$ and $P_{\text{in}}^{(l)}$ of dimension 512 (i.e., the total per-layer compression dimension is $\kappa^{(l)} = 512 \times 512$), and no second-stage projector ($P_{\text{final}}^{(l)} = I$), refreshing the subspace every 200 steps.

D.2.3 Data Regularization

Data regularization is guided by a small held-out target set of 16 samples ($n_{\text{val}} = 16$). At each training step, we form a merged batch of $n = 8$ training samples and $m = 1$ target sample (resampled per step from the held-out target pool), and score candidate training examples bycompressed scoring (Section 4.3.3) with gradient compressors of dimension $\kappa^{(l)} = 64 \times 64$ for each layer, including MeSO. Note that under MeSO, the score compression and update compression are done using different compressors (64×64 for scoring vs. 512×512 for updates), allowing a fair comparison across fine-tuning methods.

We solve S_t (or $S_{t,l}$ in the case of Layer-Wise Subset Update) by the top-k strategy, where we consider a retaining rate of 50% (i.e., $k = 4$ out of batch size $n = 8$).

D.2.4 Case Study

Figure 13 extends the score analysis from Section 5.1.3 to the three QA settings. The same qualitative pattern holds across all four settings: a single `down_proj` layer at an early block (block 0 or 1) produces scores $22\times$–$38\times$ larger than the second-largest layer type at the same block, and the global ranking is consequently dominated by that layer. Q and K projections, by contrast, retain near-zero correlation ($\rho \lesssim 0.3$) with the global subset ranking across all four settings, confirming that the score-magnitude skew is not specific to any particular general/target pair or training-pool composition.

D.3 Reinforcement Learning from Human Feedback

We now provide the missing details of the RLHF experiments in Section 5.2, including details on datasets, libraries, training configurations, and data regularization.

D.3.1 Libraries and Datasets

We implement PPO-based RLHF using `TRL`[14] [von Werra et al., 2020] (Apache-2.0 license) and sample prompts from `real-toxicity-prompts`[15] [Gehman et al., 2020] (Apache-2.0 license). For rewards, we use the `LFTW R4 Target` toxicity detector[16] [Vidgen et al., 2021], and we evaluate toxicity using `da-electra-hatespeech-detection`[17]. We note that this experimental setup largely follows the official example provided from `TRL`,[18] but with a newer `TRL` version.

D.3.2 Training Configurations

For each prompt, the policy generates a continuation of up to 30 tokens, scored by the reward model; PPO updates maximize reward while penalizing deviation from a frozen reference policy via an adaptive KL penalty. We use 4 PPO epochs with mini-batch size 4, rollout batch size 256, learning rate 10^{-5} (policy) and 5×10^{-4} (value head), linear schedule with 3% warmup, and no weight decay. LoRA is applied with rank 16, $\alpha = 32$, and dropout 0.05. The PPO surrogate uses clipping range 0.2 for both policy and value, value function coefficient 0.1, discount factor $\gamma = 1.0$, and GAE parameter $\lambda = 0.95$. The KL penalty is initialized at 0.02 and adapted via an AdaptiveKLController with horizon target 70.0, using the k_1 estimator; early stopping is enabled when the per-step KL exceeds 1.5×0.3. Evaluation is performed every PPO step by generating continuations for 500 held-out prompts (batch size 256) and reporting the mean predicted toxicity.

[14] https://huggingface.co/docs/trl/index
[15] https://huggingface.co/datasets/allenai/real-toxicity-prompts
[16] https://huggingface.co/facebook/roberta-hate-speech-dynabench-r4-target
[17] https://huggingface.co/alexandrainst/da-hatespeech-detection-base
[18] https://huggingface.co/docs/trl/v0.4.7/en/detoxifying_a_lm

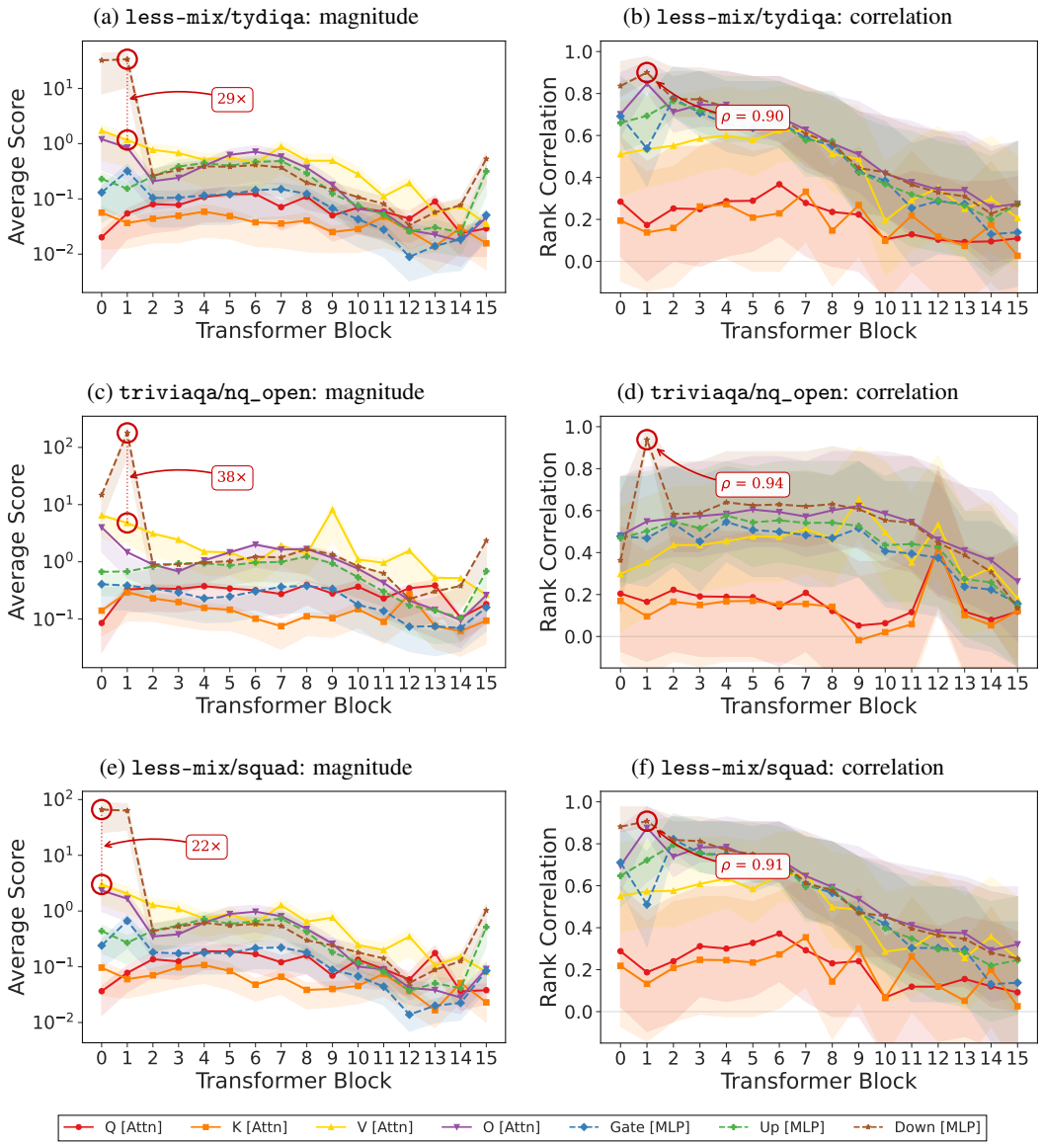

Figure 13: Per-layer scores on the three QA-only settings (full-parameter SFT trace), analogous to Figure 8. Magnitude ratios at the dominant block: $22\times$ (`less-mix/tydiqa`), $29\times$ (`triviaqa/nq_open`), $38\times$ (`less-mix/squad`). Spearman ρ of the dominant `down_proj` layer with the global ranking: 0.91, 0.94, and 0.90 respectively, while Q and K projections remain near zero ($\rho \lesssim 0.3$) in all three settings.

D.3.3 Data Regularization

In this setting, we consider exact scoring with the per-token inner product introduced in Section 4.3.2. Two types of "target set" are considered: (i) the same rollout batch (self-referencing), or (ii) new rollouts generated from a held-out target set of 1024 prompts, processed in batches of 256. Furthermore, we consider two target loss functions for the target signals: (i) reward-weighted log-probability, and (ii) the pre-clip PPO surrogate loss (i.e., the training objective before clipping).

Finally, we solve S_t (or $S_{t,l}$ in the case of Layer-Wise Subset Update) via *negative score filtering*, where at each PPO step, samples (i.e., rollouts) with negative scores are dropped from the update.

D.4 Reinforcement Learning with Verifiable Rewards

We now provide the missing details of the RLVR experiments in Section 5.3, including details on datasets, libraries, training configurations, and data regularization.

D.4.1 Libraries and Datasets

We use the `Verl` framework[19] [Sheng et al., 2024] (Apache-2.0 license) to train on the `MATH` dataset[20] [Hendrycks et al., 2021] (MIT license). We note that this experimental setup largely follows the official example provided from `Verl`,[21] with different models and datasets.

D.4.2 Training Configurations

Each training sample is a math problem; the model generates 8 candidate solutions and receives a verifiable reward from exact-match checking against the ground-truth answer. The policy is optimized using GRPO [Shao et al., 2024] with training batch size 128, mini-batch size 32, micro-batch size 2 per GPU, KL coefficient 0.001 (low-variance estimator), learning rate 10^{-6}, maximum prompt length 1024, and maximum response length 2048. Gradient checkpointing is enabled. We train for 3 epochs and evaluate every 3 steps. All other settings follow the default `Verl` configuration.

D.4.3 Data Regularization

We instantiate the data regularization framework as in-run cross-validation with again the per-token inner product introduced in Section 4.3.2: at each step, a random subset drawn from a pool of held-aside training prompts serves as the target batch, and the reward-weighted log-probability on the target rollouts provides the target signal. Then, negative score filtering is applied to the training batch based on scores with this target, dropping all samples with negative alignment scores. The target pool contains 512 prompts sampled from the training set, with target batch size 64; target rollouts are regenerated at every step using the current policy.

D.5 System Efficiency

We now provide additional benchmarking results that we omitted in Section 5.4, where specifically, we provide a comparison with the two-pass implementation of Global Subset Update (named Subset (2P)) as introduced in Section C.2. Tables 7 and 8 provide per-component breakdowns (ms) for the three benchmark models on a single `A40` GPU across two batch configurations, without and with activation checkpointing, respectively. The format mirrors Table 3; for Subset (2P), the scoring and gradient passes are shown separately. Table 9 consolidates the corresponding peak GPU memory.

[19] https://github.com/verl-project/verl

[20] https://github.com/hendrycks/math

[21] https://github.com/verl-project/verl/blob/65eb5a15058d73ef54064092222b7af67d658241/examples/grpo_trainer/run_qwen3-8b.sh

Table 7: Per-step component breakdown (ms) on a single `A40` GPU across three models and two batch configurations ($m = 1$, $k = n/2$, compressed scoring).

(a) SMOLLM2-360M, $n = 8$, $T = 512$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	78.4	88.6	88.9	88.7	46.5
Backward	155.7	234.0	193.0	161.5	81.4
`a.grad`	32.5	34.2	33.3	33.4	16.1
`scoring`	—	43.8	22.5	22.5	—
`w.grad`	27.5	43.5	30.7	—	15.5
`autograd`	95.6	112.7	106.5	105.6	49.8
Optimizer	26.6	26.6	26.6	—	27.8
Total	**261**	**349** (+34.0%)	**309** (+18.4%)	**405** (+55.2%)	

(b) SMOLLM2-360M, $n = 2$, $T = 1024$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	46.9	61.5	61.4	61.3	47.0
Backward	83.1	229.0	173.8	139.9	72.2
`a.grad`	16.1	27.0	26.6	26.7	14.2
`scoring`	—	62.7	36.8	33.6	—
`w.grad`	15.5	48.1	24.0	—	12.5
`autograd`	51.6	91.2	86.5	79.6	45.5
Optimizer	26.6	26.6	26.6	—	27.8
Total	**157**	**317** (+102.5%)	**262** (+67.2%)	**348** (+122.4%)	

(c) TINYLLAMA-1.1B, $n = 8$, $T = 512$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	155.5	168.4	167.9	168.2	82.3
Backward	312.5	352.8	344.2	270.0	163.5
`a.grad`	93.3	96.6	96.6	96.6	48.8
`scoring`	—	35.6	31.3	31.3	—
`w.grad`	91.7	77.2	73.4	—	47.6
`autograd`	127.6	143.4	142.9	142.1	67.1
Optimizer	81.2	81.3	81.2	—	81.2
Total	**549**	**602** (+9.7%)	**593** (+8.0%)	**765** (+39.3%)	

(d) TINYLLAMA-1.1B, $n = 2$, $T = 1024$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	83.8	121.7	121.7	121.6	48.8
Backward	166.5	241.3	233.5	188.9	91.8
`a.grad`	48.9	67.8	67.7	67.7	27.7
`scoring`	—	23.4	20.1	20.1	—
`w.grad`	47.5	47.5	43.7	—	26.1
`autograd`	70.1	102.6	101.9	101.1	38.0
Optimizer	81.2	81.2	81.2	—	81.2
Total	**332**	**444** (+34.0%)	**436** (+31.6%)	**532** (+60.6%)	

(e) LLAMA-3.2-3B, $n = 8$, $T = 512$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	343.3	387.5	390.0	388.9	183.1
Backward	828.2	825.6	820.2	583.5	427.3
`a.grad`	228.9	255.8	256.8	256.9	118.1
`scoring`	—	49.7	45.4	45.3	—
`w.grad`	342.2	233.1	235.4	—	174.6
`autograd`	257.1	287.1	282.6	281.4	134.6
Optimizer	235.4	235.4	235.5	—	235.5
Total	**1407**	**1449** (+3.0%)	**1446** (+2.8%)	**1818** (+29.2%)	

(f) LLAMA-3.2-3B, $n = 2$, $T = 1024$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	184.9	264.0	265.3	264.8	101.0
Backward	431.8	509.7	503.5	393.0	202.5
`a.grad`	118.0	165.4	166.3	166.0	63.7
`scoring`	—	34.6	31.4	31.4	—
`w.grad`	174.4	108.2	109.1	—	63.2
`autograd`	139.4	201.5	196.6	195.6	75.6
Optimizer	235.6	235.4	235.5	—	235.5
Total	**852**	**1009** (+18.4%)	**1004** (+17.8%)	**1197** (+40.4%)	

Table 8: Per-component breakdown (ms) with activation checkpointing on a single A40 GPU across three models and two batch configurations ($m = 1$, $k = n/2$, compressed scoring). Format mirrors Table 7.

(a) SMOLLM2-360M, $n = 8$, $T = 512$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	88.2	109.7	107.7	98.9	73.4
Backward	249.0	311.4	297.1	258.6	134.7
a.grad	40.7	41.0	40.8	40.9	20.9
scoring	—	33.4	26.6	26.5	—
w.grad	36.2	43.3	37.5	—	20.0
autograd	172.0	193.8	192.1	191.1	93.8
Optimizer	26.8	26.8	26.8	—	27.8
Total	**364**	**448** (+23.0%)	**432** (+18.6%)	**593** (+63.0%)	

(b) SMOLLM2-360M, $n = 2$, $T = 1024$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	63.9	77.0	77.1	69.6	72.3
Backward	135.4	305.6	252.8	218.8	148.0
a.grad	20.8	34.5	34.1	34.1	18.5
scoring	—	61.9	39.6	39.0	—
w.grad	20.0	52.4	25.7	—	15.3
autograd	94.6	156.8	153.3	145.7	114.2
Optimizer	26.9	26.8	26.8	—	28.0
Total	**226**	**409** (+81.0%)	**357** (+57.8%)	**537** (+137.3%)	

(c) TINYLLAMA-1.1B, $n = 8$, $T = 512$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	154.9	167.4	167.1	174.2	82.2
Backward	453.6	504.3	495.9	421.8	238.3
a.grad	93.3	96.8	96.7	96.8	48.9
scoring	—	35.5	31.3	31.3	—
w.grad	91.8	77.1	73.4	—	47.6
autograd	268.5	294.9	294.5	293.6	141.7
Optimizer	81.2	81.3	81.3	—	81.2
Total	**690**	**753** (+9.2%)	**744** (+7.9%)	**998** (+44.7%)	

(d) TINYLLAMA-1.1B, $n = 2$, $T = 1024$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	84.0	121.1	121.0	128.9	50.2
Backward	242.8	351.5	343.8	299.3	136.9
a.grad	49.0	67.8	67.8	67.8	27.7
scoring	—	23.3	20.2	20.0	—
w.grad	47.6	47.4	43.6	—	26.1
autograd	146.3	213.0	212.2	211.4	83.1
Optimizer	81.3	81.2	81.2	—	81.2
Total	**408**	**554** (+35.7%)	**546** (+33.8%)	**697** (+70.7%)	

(e) LLAMA-3.2-3B, $n = 8$, $T = 512$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	389.2	437.4	438.4	437.9	216.5
Backward	1233.5	1279.1	1267.8	1012.7	645.3
a.grad	269.4	299.4	299.3	299.3	136.0
scoring	—	60.2	54.7	54.8	—
w.grad	371.6	254.9	254.0	—	189.5
autograd	592.5	664.6	659.7	658.6	319.8
Optimizer	236.8	236.8	236.8	—	236.8
Total	**1860**	**1953** (+5.0%)	**1943** (+4.5%)	**2549** (+37.1%)	

(f) LLAMA-3.2-3B, $n = 2$, $T = 1024$.

Component	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	
				Score	Grad
Forward	214.3	303.2	302.4	302.5	125.9
Backward	653.6	829.3	819.4	693.2	331.1
a.grad	136.0	198.4	198.3	198.4	75.0
scoring	—	38.0	33.8	33.7	—
w.grad	189.5	125.6	124.9	—	76.2
autograd	328.1	467.4	462.3	461.1	179.8
Optimizer	236.8	236.8	236.8	—	236.8
Total	**1105**	**1369** (+24.0%)	**1359** (+23.0%)	**1689** (+52.9%)	

Table 9: Peak GPU memory (GB) across three models and two batch configurations ($m = 1$, $k = n/2$, compressed scoring) on a single A40 GPU, with and without activation checkpointing (ckpt).

Model	Setting	$n = 8$, $T = 512$				$n = 2$, $T = 1024$			
		Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)	Full-Training	Layer-Wise Subset	Global Subset (1P)	Global Subset (2P)
SMOLLM2-360M	No ckpt	9.4	10.3	10.4	10.4	5.7	7.7	7.8	7.7
	With ckpt	4.6	4.9	7.5	4.9	3.4	4.0	6.0	4.0
TINYLLAMA-1.1B	No ckpt	15.0	16.2	16.2	16.2	10.6	13.0	13.0	13.0
	With ckpt	10.3	10.4	14.5	10.4	10.3	10.4	12.5	10.4
LLAMA-3.2-3B	No ckpt	37.9	40.7	40.7	40.7	29.9	33.2	33.9	33.2
	With ckpt	29.9	30.1	38.1	30.1	29.9	30.1	33.9	30.1