DATTRI-LLM: A UNIFIED AND EFFICIENT LIBRARY FOR TRAINING DATA ATTRIBUTION AT LLM SCALE

Anonymous authors
Paper under double-blind review

ABSTRACT

Training data attribution (TDA) estimates the contribution of individual training examples to model outputs. Most scalable TDA methods rely on per-example gradients, whose computation and use at LLM scale pose challenges in efficiency, compatibility, and extensibility. We introduce `dattri-LLM`, a TDA library that makes gradient-based attribution more practical at scale. For efficiency, `dattri-LLM` uses compact gradient representations and dynamically routes gradient operations based on a cost model. For compatibility, its capture mechanism collects per-example gradients from existing training loops that call `backward()`, without requiring changes to the loop or its configuration. This includes distributed training with DDP and FSDP and pipelines built with HuggingFace Transformers, TRL, and OLMo. For extensibility, `dattri-LLM` exposes reusable gradient operations and training-time callbacks for implementing attribution methods and applications. These interfaces support a variety of attribution methods, including gradient similarity, curvature-based influence, and trajectory-based methods, as well as applications that act on gradients during training, such as online data selection. On the same hardware and workload, `dattri-LLM` achieves 3.2$\times$ the throughput of the fastest competing library on average, scales multiple attribution methods to 110B-parameter models across four H200 GPUs, and offers superior attribution fidelity–cost trade-offs across a range of models with different model families and scales.

1 INTRODUCTION

Training data attribution (TDA) (Deng et al., 2025) estimates the contribution of individual training examples to a model's outputs, with applications in model debugging (Koh & Liang, 2017; Yeh et al., 2018; Kwon et al., 2024), data selection (Xia et al., 2024; Wang et al., 2024; Hu et al., 2026), and data valuation (Ghorbani & Zou, 2019; Jia et al., 2019; Kwon & Zou, 2022). Many scalable TDA methods compare gradients of a query objective with per-example training gradients, either directly, as in TracIn (Pruthi et al., 2020), or after transformations such as projection and preconditioning, as in TRAK and EK-FAC-based influence estimation (Park et al., 2023; Grosse et al., 2023). Although their formulations differ, these methods share computational building blocks: obtaining per-example gradients and applying operations such as projection, preconditioning, and inner products.

At LLM scale, these computations pose three systems challenges. The first is *efficiency*. A single per-example gradient can contain billions of entries, making its explicit construction and storage expensive (Schioppa et al., 2022; Grosse et al., 2023; Park et al., 2023). Structured representations can reduce this cost (Li et al., 2022), but their relative efficiency depends on the layer shape and the operation being performed. Efficient execution therefore requires choosing suitable representations for different gradient operations. The second is *compatibility*. Existing libraries (Deng et al., 2024; Quirke et al., 2026) offer task-based attribution workflows, trainer integrations, and custom-loop gradient collectors. However, using these capabilities within an existing training pipeline can still require changes to the training code or framework-specific integration. The third is *extensibility*. Implementing an attribution method requires reusable components, such as gradient capture, projection, and curvature estimation. Applications that act during training additionally need defined callback points at which they can access captured gradients and inspect or modify training state. Supporting both requires interfaces that separate method-specific logic from more general modules, such as gradient capture and execution.

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

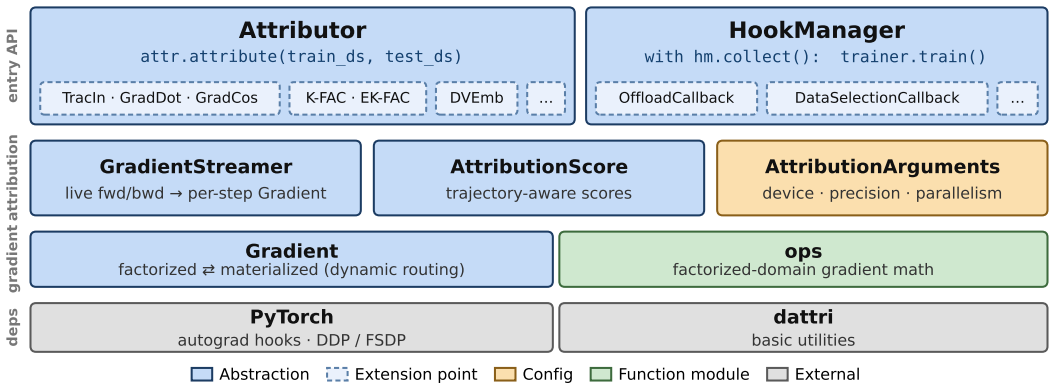

Figure 1: **Architecture overview of `dattri-LLM`.** From bottom to top, the stack comprises external dependencies (gray), per-example gradient representations and operations (green), shared attribution components, and the entry API. Solid blue boxes denote classes that users instantiate and call; orange denotes configuration. Dashed boxes denote abstract base classes that serve as extension points: subclassing `Attributor` adds an attribution method, while subclassing `HookManagerCallback` adds an application that acts during gradient capture.

To close these gaps, we introduce `dattri-LLM`, a library for **Da**ta **Attri**bution at **LLM** scale, built around a unified gradient interface. Its design addresses these three challenges:

Efficiency: exact gradient representations with FLOP-aware routing. `dattri-LLM` supports two exact representations of per-example gradients for supported layers: *factorized* and *materialized*. A FLOP-aware cost model selects between them for each layer and operation, reducing execution cost without introducing additional approximation into the attribution computation. Shared gradient operations make this routing available across attribution methods (§3.2). Across the methods and projection regimes evaluated in §4.1, `dattri-LLM` achieves an average speedup of 3.2× over the fastest competing library for each configuration.

Compatibility: non-invasive gradient capture from existing training loops. Alongside an end-to-end attribution interface, `dattri-LLM` provides an *open-loop* interface that uses autograd hooks to capture per-example gradients during backward passes in existing training loops. Users enable capture by enclosing the training call in a context manager, without editing the training loop, optimizer, or trainer. We demonstrate this integration with HuggingFace `Trainer` (Wolf et al., 2020), TRL (von Werra et al., 2020), and OLMo (Groeneveld et al., 2024), and evaluate capture under DDP (Li et al., 2020) and FSDP (Zhao et al., 2023) (§3.3). Captured gradients can also be cached and reused across queries and attribution methods when the cache satisfies their gradient and checkpoint requirements, avoiding repeated training-side computation.

Extensibility: interfaces for new methods and training-time applications. `dattri-LLM` exposes an `Attributor` interface for implementing gradient-based attribution methods using a common gradient stream and shared gradient operations. Its `HookManagerCallback` interface provides callback points during gradient capture, allowing applications to inspect or modify training state using the captured gradients (§3.4). We demonstrate this capability by integrating a recent online data-selection method into an existing fine-tuning loop through a callback (§5).

2 Related Work

We review TDA methods, whose shared building blocks motivate the design of `dattri-LLM`, and existing TDA libraries, against which we position this library.

TDA methods. TDA methods estimate how individual training examples contribute to a model's behavior, and are commonly grouped by the principles underlying their influence measures. *Influence-function-based* methods (Koh & Liang, 2017) use local approximations to estimate the

Table 1: Comparison of recent libraries for LLM-scale TDA. ✓/✗ indicates supported / not supported; ∼ indicates partial support: Kronfluence uses a cost model, but it always builds full gradients on the query side and optionally builds training-side gradients on demand. Bergson integrates with a wrapped HuggingFace `Trainer` through callbacks that record gradients without acting on the training state. LogIX supports DDP but not FSDP. Kronfluence and Bergson provide interfaces for customizing specific steps of attribution, such as gradient collection or preconditioning, but not for defining a new end-to-end attribution method.

	`dattri-LLM`	LogIX	Kronfluence	Bergson
Efficiency				
Dimension reduction	✓	✓	✗	✓
Training-gradient reuse	✓	✓	✗	✓
Cost-based operation routing	✓	✗	✓	✗
Full gradients built on demand	✓	✗	∼	✗
Compatibility				
Integration with existing training	✓	✗	✗	∼
Distributed execution	✓	∼	✓	✓
Extensibility				
Interface for new attribution methods	✓	✗	∼	∼
Interface for training-time applications	✓	✗	✗	✗

effect of removing training examples. Scalable variants use Arnoldi iteration (Schioppa et al., 2022), Kronecker-factored curvature (K-FAC and EK-FAC) (Martens & Grosse, 2015; George et al., 2018; Grosse et al., 2023), or closed-form inverse-curvature approximations such as DataInf (Kwon et al., 2024). TRAK (Park et al., 2023) and LoGRA (Choe et al., 2025) reduce computational costs through gradient projection. *Weighted marginal contribution* methods, such as Data Shapley (Ghorbani & Zou, 2019), average an example's marginal contribution across training subsets under a specified weighting. *Training-dynamics-based* methods use information along the optimization trajectory. TracIn (Pruthi et al., 2020) accumulates gradient inner products over checkpoints, while Grad-Dot and Grad-Cos (Charpiat et al., 2019) evaluate gradient similarity at a single checkpoint. In-Run Data Shapley (Wang et al., 2025a) assigns contributions within a realized training run, while data value embeddings (Wang et al., 2025b) and SOURCE (Bae et al., 2024) approximate how an example's influence propagates through subsequent training updates. AdamW-influence (Deng et al., 2026) extends trajectory-based influence estimation to account for AdamW's first- and second-moment states, adaptive scaling, and weight decay. *Simulator-based* methods, including Datamodels (Ilyas et al., 2022) and Simfluence (Guu et al., 2023), fit surrogate models to observations from training runs to predict model behavior under different training subsets or curricula. Other approaches include representer points (Yeh et al., 2018), which decompose predictions using weighted similarities between learned features. Across these families, gradient-based estimators support applications such as data selection with LESS (Xia et al., 2024) and pretraining attribution with TrackStar (Chang et al., 2025), while techniques such as gradient sparsification in GraSS (Hu et al., 2025) improve scalability. Despite their differences, these methods share the same building blocks of per-example gradients, projection or preconditioning, and pairwise scoring, which is the layer at which `dattri-LLM` is organized. We refer readers to Deng et al. (2025) for a comprehensive review.

TDA libraries. General-purpose TDA libraries such as `dattri` (Deng et al., 2024), `pyDVL` (TransferLab Team, 2024), OpenDataVal (Jiang et al., 2023), and Influenciae (Picard et al., 2024) support data valuation and attribution, including game-theoretic and influence-function estimators. However, most are designed for smaller machine learning models that predate modern LLMs. Among them, `dattri` provides projected attributors that scale to billion-parameter models on a single GPU, but does not support attribution for models with tens or hundreds of billions of parameters across multiple GPUs. Libraries specifically developed for LLM-scale attribution include Kronfluence (Grosse et al., 2023), which implements K-FAC and EK-FAC influence functions with distributed and token-level scoring; LogIX (Choe et al., 2025), which captures projected gradients through patched HuggingFace or lightning trainers; and Bergson (Quirke et al., 2026), which supports collecting projected or full gradients into a store and integrates with the HuggingFace `Trainer`. Table 1 compares these libraries with `dattri-LLM` along the three design principles of §3 in terms of efficiency-related capabilities, compatibility with existing training pipelines, and

interfaces for extension. Across these rows, `dattri-LLM` covers the efficiency, compatibility, and extensibility requirements together.

3 DESIGN

In this section we introduce the design principles of `dattri-LLM`. We first give an overview of its architecture (§3.1), and then describe how each of the three properties from §1 is realized: efficiency through the choice of gradient representation (§3.2), compatibility through non-invasive gradient capture (§3.3), and extensibility through two designated extension points (§3.4). Implementation details are given in Appendix A.

3.1 ARCHITECTURE OVERVIEW

`dattri-LLM` is organized into four tiers (Figure 1). The bottom tier holds the dependencies: PyTorch (Paszke et al., 2019), whose autograd hooks and DDP/FSDP wrappers the library builds on, and `dattri` (Deng et al., 2024), from which it imports basic utilities such as random projectors. The *gradient* tier defines the `Gradient` object, which stores a batch of per-example gradients across selected layers, using either *factorized* or *materialized* representation that we will introduce in §3.2, together with the mathematical operations on it. The *attribution* tier holds the logic shared by all attribution methods: the `GradientStreamer` that produces per-step gradients from a model, the `AttributionScore` result container, the `AttributionArguments` configuration, and the callbacks that act on the capture loop. The top tier is the entry API, which offers two ways in. `Attributor` is the *closed-loop* interface: given a model, training set, and query set, it directly returns scores for pre-built attribution methods, including TracIn (Pruthi et al., 2020), LESS (Xia et al., 2024), K-FAC and EK-FAC influence functions (Martens & Grosse, 2015; George et al., 2018; Grosse et al., 2023), data-value embeddings (Wang et al., 2025b) and AdamW-Influence (Deng et al., 2026). `HookManager` is the *open-loop* interface: it captures per-example gradients from `.backward()` executed inside a `with` block, so attribution can be attached to a training loop the user already owns (§3.3).

3.2 EFFICIENCY: EXACT GRADIENT REPRESENTATIONS WITH FLOP-AWARE ROUTING

Computing and comparing per-example gradients is a major cost of TDA at LLM scale. `dattri-LLM` exploits the fact that the same gradient admits two exact representations whose computational costs differ across operations and layer shapes. It selects the representation separately for each layer and operation using a FLOP-aware cost model.

Two representations of gradients. Consider a linear layer $z = Wa$ with $W \in \mathbb{R}^{N_o \times N_i}$, per-example input $a \in \mathbb{R}^{N_i \times T}$ over T token positions, and output $z \in \mathbb{R}^{N_o \times T}$. Let D denote the differentiation operator $\partial\ell/\partial(\cdot)$ and $g = \mathrm{D}z \in \mathbb{R}^{N_o \times T}$ denote the gradient of the per-example loss ℓ with respect to the layer output. By the chain rule, the weight gradient is $\mathrm{D}W = \partial\ell/\partial W = g\,a^\top$, or, in flattened form,

$$\gamma := \mathrm{vec}(\mathrm{D}W) = \sum_{t=1}^{T} a_t \otimes g_t, \tag{1}$$

where a_t and g_t are the columns at token position t, $\mathrm{vec}(\cdot)$ stacks matrix columns, and $\otimes$ denotes the Kronecker product. We call the dense matrix $\mathrm{D}W$ the *materialized* representation and the pair (a, g) the *factorized* representation. The materialized gradient can be recovered exactly from the factors. The factorized form stores $T(N_i + N_o)$ values, compared with $N_i N_o$ for the materialized form. It therefore uses less storage when $T < N_i N_o/(N_i + N_o)$, favoring wide layers and short sequences. Convolutional and embedding layers also admit related factorizations (Appendix A.1), extending this representation beyond linear layers.

Gradient postprocessing. `dattri-LLM` supports optional transforms before scoring, including *dimension reduction* and *optimizer preconditioning*. Dimension reduction can operate on either the factors or the materialized gradient. *Factor projection* applies random matrices $P_a \in \mathbb{R}^{k_a \times N_i}$ and

$P_g \in \mathbb{R}^{k_g \times N_o}$ to the two factors:

$$\tilde{a}_t = P_a a_t, \qquad \tilde{g}_t = P_g g_t, \qquad \sum_{t=1}^{T} \tilde{a}_t \otimes \tilde{g}_t = (P_a \otimes P_g)\gamma, \tag{2}$$

where k_a and k_g are the projected dimensions. This preserves the factorized structure in Eq. 1, allowing subsequent operations to use the same formulas in the projected space (Choe et al., 2025; Hu et al., 2025). *Materialized projection* instead applies a single random matrix $P \in \mathbb{R}^{k_{\text{dense}} \times (N_i N_o)}$ to the flattened gradient, yielding $\tilde{\gamma} = P\gamma$, where k_{dense} is the projection dimension for the dense gradient. This approach, used in TRAK (Park et al., 2023), also applies to layers without a factorized gradient representation. Supported random projections include Gaussian (Johnson & Lindenstrauss, 1984), Rademacher (Achlioptas, 2003), sparse Johnson–Lindenstrauss transforms (Kane & Nelson, 2014), and the sparsified projection of GraSS (Hu et al., 2025). Alternatively, a random mask selects a subset of gradient entries, which can be computed directly from the factors without materializing the full gradient (Appendix A.2).

Optimizer preconditioning transforms a per-example gradient into an update direction using the training optimizer's current state. For Adam (Kingma & Ba, 2015), the direction for sample i is

$$\Gamma^{(i)} = \frac{\hat{m}^{(i)}}{\sqrt{\hat{v}^{(i)}} + \epsilon}, \qquad m^{(i)} = \beta_1 m + (1-\beta_1)\gamma^{(i)}, \qquad v^{(i)} = \beta_2 v + (1-\beta_2)(\gamma^{(i)})^2, \tag{3}$$

where parenthesized superscripts index samples, (m, v) are the optimizer's moment estimates for the corresponding parameters before the current step, β_1, β_2 are their decay rates, and $\epsilon > 0$ ensures numerical stability. At optimizer step τ, the bias-corrected updated moments are $\hat{m}^{(i)} = m^{(i)}/(1-\beta_1^\tau)$ and $\hat{v}^{(i)} = v^{(i)}/(1-\beta_2^\tau)$ (Appendix A.3). This transform enables optimizer-aware methods such as LESS (Xia et al., 2024) and AdamW-influence (Deng et al., 2026).

Factorized inner product. For samples i and j with T_i and T_j token positions, respectively, the Kronecker-product inner-product identity gives

$$\left\langle \gamma^{(i)}, \gamma^{(j)} \right\rangle = \left\langle \mathrm{D}W^{(i)}, \mathrm{D}W^{(j)} \right\rangle_F = \sum_{t=1}^{T_i} \sum_{s=1}^{T_j} \left\langle a_t^{(i)}, a_s^{(j)} \right\rangle \left\langle g_t^{(i)}, g_s^{(j)} \right\rangle, \tag{4}$$

where $\langle \cdot, \cdot \rangle_F$ denotes the Frobenius inner product. Thus, exact gradient inner products can be computed directly from the factors without materializing the weight gradients. Factorized representations also support the preconditioned gradient comparisons used in K-FAC (Martens & Grosse, 2015) and EK-FAC (George et al., 2018). Retaining the token axis additionally enables attribution at token granularity.

FLOP-aware routing. Eq. 4 compares all pairs of token positions, so its cost is quadratic in sequence length. Alternatively, materializing the gradients incurs a construction cost that can be amortized over subsequent dense inner products. For a cross-Gram matrix between batches of B_1 and B_2 examples, each with T token positions, the factorized and materialized routes cost $O\big(B_1 B_2 T^2 (N_i + N_o)\big)$ and $O((B_1 + B_2) T N_i N_o + B_1 B_2 N_i N_o)$, respectively. Neither route is uniformly cheaper: short sequences and wide layers favor factorized computation, while long sequences and large comparison batches favor materialization. Factor projection replaces N_i, N_o with k_a, k_g in these costs and can change which route is preferable. `dattri-LLM` therefore estimates the FLOP cost of each route per layer and operation and selects the cheaper one (Appendix A.4). An ablation in Appendix B.4 evaluates the benefit of this routing strategy.

3.3 Compatibility: non-invasive gradient capture from arbitrary training loops

In `dattri-LLM`, attribution can be integrated into an existing pipeline without modifying its training loop. Specifically, the *open-loop* interface `HookManager` registers autograd hooks on selected layers and captures the two factors in Eq. 1 during each backward pass triggered by `.backward()`. This mechanism applies uniformly to pretraining, supervised fine-tuning, and reinforcement-learning loops. For example, unmodified HuggingFace `Trainer` (Wolf et al., 2020),

TRL (von Werra et al., 2020), and OLMo (Groeneveld et al., 2024) pipelines can be instrumented simply by wrapping `trainer.train()` with `with hm.collect()`. The design also naturally accommodates distributed training frameworks such as DDP (Li et al., 2020) and FSDP (Zhao et al., 2023). Captured gradients may optionally be cached, allowing the same attribution code to score them either online or later from a store persisted during an ordinary training run. Appendix A.5 details the mechanisms underlying these capabilities.

3.4 Extensibility: pluggable interfaces over a single capture path

`dattri-LLM` provides two extension points. The `Attributor` abstraction supports the implementation of new gradient-based attribution methods. `GradientStreamer` exposes a uniform, well-defined interface to collect per-example gradients. The gradient tier provides the operations required by attribution methods, including inner products, norms, projections, and Kronecker-factored statistics, with each operation routed as described in §3.2. A new method can therefore be implemented as a single subclass that consumes these modules and produces the desired attribution scores. The built-in TracIn (Pruthi et al., 2020), K-FAC (Martens & Grosse, 2015), EK-FAC (George et al., 2018; Grosse et al., 2023), and data-value-embedding (Wang et al., 2025b) attributors are implemented through this same interface.

The second extension point, `HookManagerCallback`, supports downstream applications that operate on gradients during training. A callback receives the capture events emitted at each step and may inspect or modify the training state. Built-in examples include an offloading callback that persists gradients to disk, and a data-selection callback that scores the current batch against a target gradient, and then removes the contributions of low-scoring examples from the parameter gradients before the optimizer step. As demonstrated in §5, this interface reduces the integration of a recent online data-selection method to only a few lines of user code.

4 Attributing LLMs at scale with high efficiency

In this section, we empirically evaluate the efficiency of `dattri-LLM` through throughput and memory scaling, and demonstrate the favorable fidelity–cost trade-offs provided by gradient-based attribution methods as model size increases. Supplementary comparisons at a fixed model size and batch size appear in Appendix B.3.

4.1 Scaling attribution with model size

Setup. We compare `dattri-LLM`, LogIX (Choe et al., 2025), Bergson (Quirke et al., 2026), and Kronfluence (Grosse et al., 2023) on GradDot, K-FAC, and EK-FAC, using Qwen models from 0.5B to 110B parameters and 512-token WikiText-103 sequences (Merity et al., 2017) in `bf16`. We use factor projection with $k_a = k_g = 64$ where supported. Throughput is measured on four H200 GPUs (141 GB each), using each library's supported distributed mode and the largest per-device batch that completes within the benchmark limits. Each run attributes 512 training sequences, and we use the largest power-of-two batch size that fits in GPU memory. Detailed configurations and timing protocols are reported in Appendix B.1.

Results. Figure 2 (top) shows that `dattri-LLM` consistently improves attribution throughput across model scales. For GradDot, it achieves 2.7–3.5× the throughput of Kronfluence and 1.4–8.9× that of Bergson across their respective completed scales. The advantage also persists when compression and batch size are matched: for K-FAC and EK-FAC, `dattri-LLM` achieves on average 3.8× the throughput of LogIX from 0.5B to 32B, and up to 6.8× at 0.5B. The larger gains over Bergson and Kronfluence on curvature-based methods additionally reflect the cost of their full-dimensional curvature factors. Beyond improving throughput, `dattri-LLM` extends the range of feasible workloads: it supports all three methods through 110B, whereas the evaluated alternatives either exceed the resource budget or lack the sharded execution needed at larger scales.

To understand the memory requirements underlying these scaling results, we also evaluate attribution at batch size one, minimizing batch-dependent memory costs and highlighting the footprint of the model and attribution computation. Figure 2 (bottom) illustrates how the gradient-processing strategy affects scalability. The benefit is most pronounced for curvature-based attribution: at 3B,

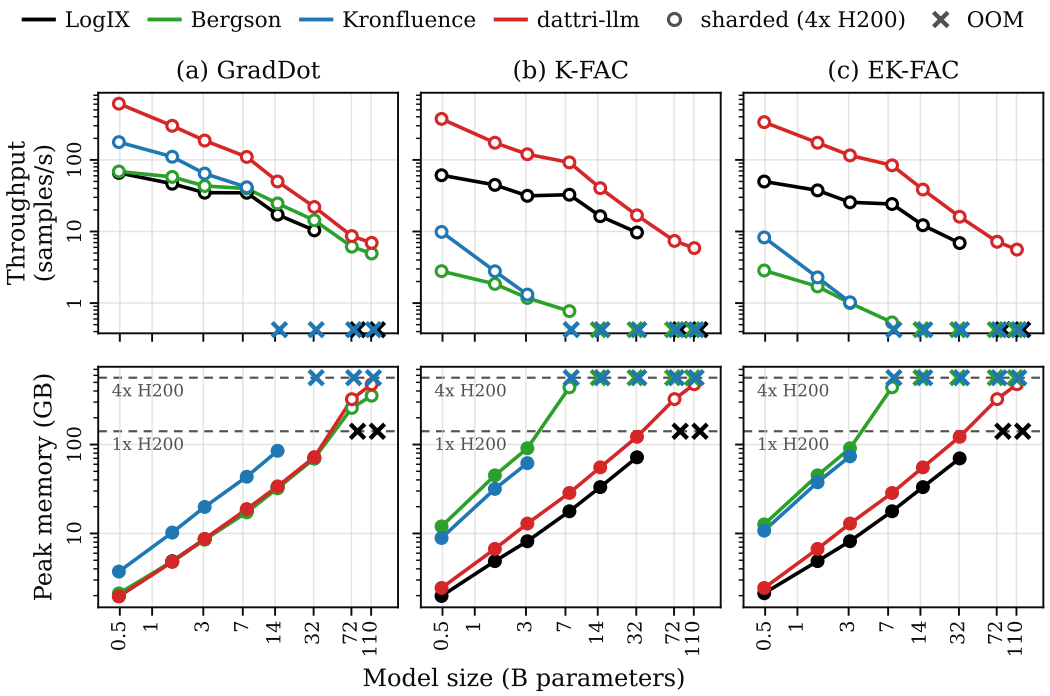

Figure 2: **Throughput and memory scaling on the Qwen family. Top:** throughput on four H200s at each configuration's largest batch that completes within the benchmark limits. Crosses indicate configurations reported as out of memory. **Bottom:** GPU memory at batch size one. Filled markers denote single-H200 runs; hollow markers denote four-H200 sharded runs, with reported aggregate memory estimates. Dashed lines indicate aggregate device capacities, and crosses mark the total capacity at which an out-of-memory failure occurs.

`dattri-LLM` uses approximately 17.5–20.9% of Kronfluence's GPU memory and 14.2% of Bergson's for K-FAC and EK-FAC. Reducing the attribution-specific memory footprint leaves more capacity for the model and enables larger workloads within the available device memory. Together with sharded execution, this allows `dattri-LLM` to retain support for curvature-based methods at scales where the full-dimensional baseline configurations become infeasible.

4.2 Attribution Fidelity and Computational Cost

The gradient-based methods supported by `dattri-LLM` approximate the effect of training-data perturbations through first-order sensitivity. We therefore complement the efficiency evaluation with counterfactual retraining, examining whether these approximations provide useful attribution at a practical computational cost.

Setup. We evaluate the fidelity and computational cost of attribution methods on training trajectories of GPT-2 (124M), Qwen2.5-1.5B, and OLMo-3-7B (Groeneveld et al., 2024). For each model, we construct 512 training sequences from WikiText-2 using the model's tokenizer, with 128 tokens per sequence, and continually pretrain all parameters for one epoch using AdamW with batch size 32. OLMo-3-7B is trained through the unmodified OLMo-core trainer, with `dattri-LLM` capturing gradients by wrapping its `fit()` call. We assess fidelity against trajectory-specific leave-one-out retraining (TSLOO) (Wang et al., 2025b). For each of 50 randomly selected training sequences, we replay training with that sequence removed and measure the resulting loss changes on 64 held-out validation sequences. For each validation query, we compute the Spearman correlation between attribution scores and these loss changes across the 50 training sequences, then average over queries. Training and evaluation use `fp32` with `TF32` and dropout disabled to resolve the small counterfactual loss changes.

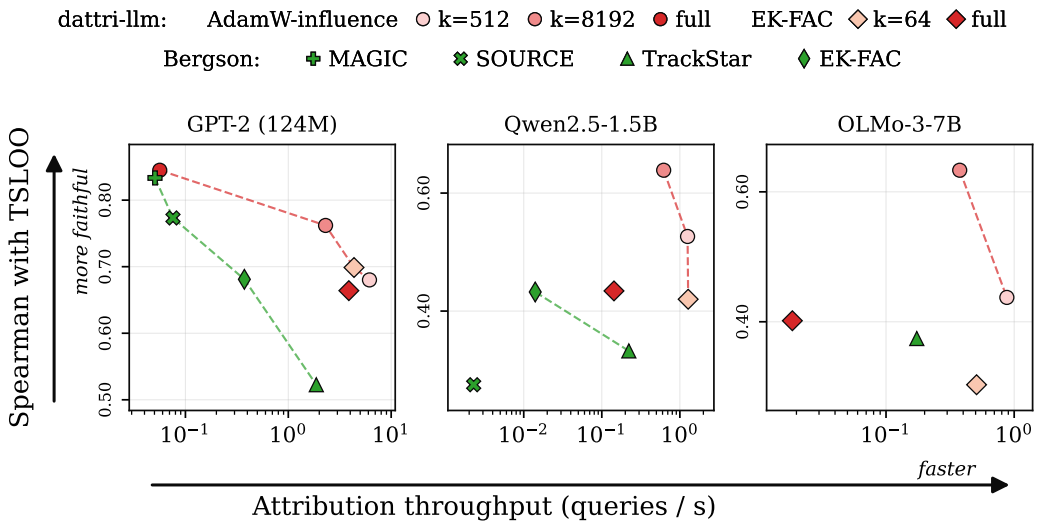

Figure 3: **Attribution fidelity versus throughput.** Higher and farther right is better. Fidelity is measured as the mean query-wise Spearman correlation with TSLOO over 64 validation queries. Throughput is 64 divided by the attribution time in seconds on one B200. Dashed lines show the empirical Pareto frontier for each library. Configurations exceeding the resource budget are omitted.

We compare AdamW-influence (Deng et al., 2026) and EK-FAC implemented in `dattri-LLM` with Bergson's MAGIC (Ilyas & Engstrom, 2025), SOURCE (Bae et al., 2024), TrackStar (Chang et al., 2025), and EK-FAC (Grosse et al., 2023). For AdamW-influence, we evaluate full-dimensional gradients and reduced representations that average scores from ten disjoint random masks of $k \in \{512, 8192\}$ gradient coordinates per layer. For EK-FAC, we evaluate full-dimensional factors and projected factors with $k_a = k_g = 64$. Each execution receives one B200, 256 GB of host memory, and 1 TB of local disk. Attribution throughput measures the number of queries scored per second, excluding trajectory training, model and data loading, and TSLOO reference construction. Appendix B.2 provides complete settings, results, and resource limits.

Results. Figure 3 shows that `dattri-LLM` offers a favorable fidelity–cost trade-off across all three models, with its configurations forming the empirical Pareto frontier among completed runs in two of the three panels. On GPT-2, full-dimensional AdamW-influence achieves a correlation of 0.845, compared with MAGIC's 0.833, while requiring less time. Masking and projection provide further control over this trade-off: for AdamW-influence, increasing the mask size k from 512 to 8192 improves correlation across all three models at the cost of approximately 2–3$\times$ longer attribution time. Beyond attribution efficiency, `dattri-LLM`'s non-invasive gradient capture enables end-to-end attribution for OLMo-3-7B trained with the unmodified OLMo-core trainer, demonstrating compatibility with an existing training pipeline at larger scale.

5 Integrating and Extending Attribution Workflows

We further demonstrate how the interfaces of §3.4 make a downstream application easy to build, highlighting the compatibility and extensibility of `dattri-LLM`. We take the most recent attribution-based data selection technique for post-training (Hu et al., 2026) and deploy it inside a minimal fine-tuning loop. Specifically, at each training step we compute GradDot scores between the samples in the current batch and a held-out validation set, and drop the lowest-scoring samples from the gradient update.

Setup. Following Hu et al. (2026), we fine-tune Llama-3.2-1B (Grattafiori et al., 2024) on Alpaca (Taori et al., 2023) with SAMSum (Gliwa et al., 2019) as the target task, dropping the bottom half of each batch at every step, and compare three variants: *Full-Training*, which trains without data selection; *Global Subset*, which ranks samples by GradDot scores aggregated over all layers; and *Layer-Wise Subset*, which ranks samples independently for each layer. All variants are run

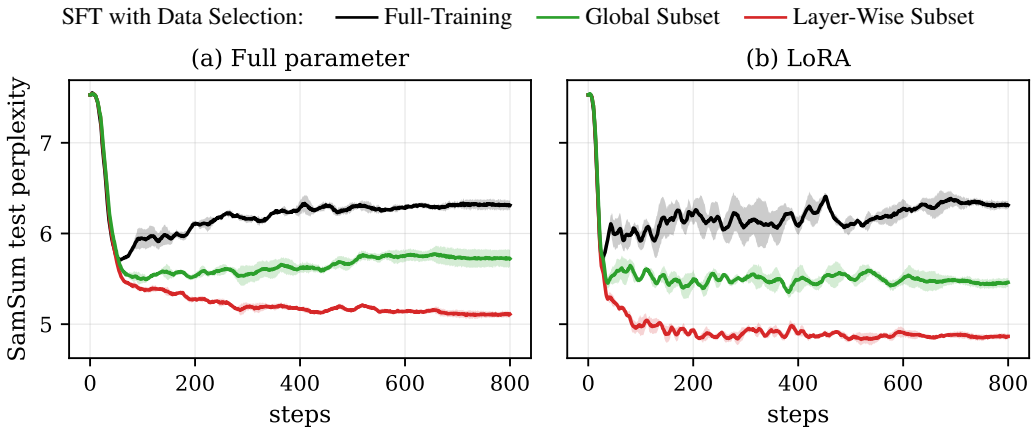

Figure 4: Attribution-guided data selection under **(a)** full-parameter and **(b)** LoRA fine-tuning (SAMSum test perplexity, lower is better; mean over 3 seeds, band ± 1 std). Both selection variants improve on training with all data, with per-layer selection best in both regimes.

under both full-parameter and LoRA (Hu et al., 2022) fine-tuning. Hyperparameters are reported in Appendix B.7. Figure 5 shows the core integration, which reduces to three steps. First, the selection policy is expressed as a `DataSelectionCallback` (§3.4), which specifies the target loader and the selection rule. Second, the callback is attached to a `HookManager`, which captures per-sample gradients during the forward and backward passes at almost no additional cost. Third, the user's training loop is enclosed in `hm.collect()`, with the loop itself unchanged. Between the backward pass and the optimizer step, the callback scores the batch and removes the dropped samples' contributions from the parameter gradients. The same procedure applies to LoRA fine-tuning, which differs from full-parameter fine-tuning only in a layer selector that restricts the hooks to the adapters.

Results. Figure 4 shows the test perplexity curves. Both selection variants outperform Full-Training in both regimes, and Layer-Wise Subset performs best, reaching a final target perplexity of 5.11 vs. 6.31 under full-parameter fine-tuning and 4.87 vs. 6.31 under LoRA. Full-Training overfits the off-target source data and its perplexity drifts upward, whereas selection acts as a regularizer that steers each update toward target-aligned examples, and the finer-grained per-layer selection regularizes further. The result reproduces the finding of Hu et al. (2026) using the integration of Figure 5 in place of a dedicated training pipeline. Notably, the `dattri-LLM` implementation incurs negligible runtime overhead compared with the original implementation by Hu et al. (2026). We report runtime measurements in Appendix B.7.

```python
from dattri_llm import DataSelectionCallback,
    HookManager
# 1. Define the selection policy
select = DataSelectionCallback(
    model, target="val_loader",
    val_loader=target_loader, val_loss_fn=loss_fn,
    selection_kwargs={"threshold": 0.5, "
     threshold_mode": "bottom_fraction"})
# 2. Define the capture hook
hm = HookManager(model, callbacks=[select])
# 3. Perform data selection
with hm.collect():
    for batch in train_loader:
        loss_fn(model, batch).backward()
        optimizer.step(); optimizer.zero_grad()
```

Figure 5: Online data selection with `dattri-LLM`.

6 Conclusion

We presented `dattri-LLM`, a library for training data attribution at LLM scale, designed for efficiency, compatibility, and extensibility. For efficiency, it represents per-example gradients exactly in factorized or materialized form and uses a FLOP-aware cost model to select the cheaper route, sharing kernels across attribution methods and supported layers without changing scores. For compatibility, non-invasive autograd hooks capture gradients from unmodified training pipelines, including DDP and FSDP. For extensibility, a callback interface supports applications such as online data selection without modifying the core. Empirically, `dattri-LLM` is $3.2\times$ faster than the fastest existing libraries on average. We believe `dattri-LLM` can lower the systems barrier to studying how training data shapes large language models.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods. We have not used generative AI tools to help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, assist with translation, clean and reformat dataset, or interpret results, and Generate synthetic data sets and support qualitative and thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to create or modify scientific figures or images, draft parts of a research paper, edit a research paper to improve readability, and format references. We have reviewed all AI-assisted work. The authors reviewed AI-assisted implementations, figures, text, and references for correctness and consistency with the methods and experimental results. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We document the library design and experimental procedures to support reproducibility. All experimental details are included in Appendix B. Appendix B.1 specifies the detailed setup for the throughput measurement experiment, including the hardware, model checkpoints, workloads, distributed configurations, batch-size selection, and timing and memory measurement protocols. Appendix B.2 details the fidelity evaluation, including training settings, counterfactual retraining, numerical controls, and evaluation metrics. Appendices B.3, B.4, and B.5 provide the settings for matched-batch comparisons, routing ablations, and token-level attribution examples. Appendix B.7 documents the datasets, hyperparameters, selection procedure, and evaluation protocol for online data selection, with results reported over three random seeds. We will officially release the code upon acceptance of the paper.

REFERENCES

Dimitris Achlioptas. Database-friendly random projections: Johnson–Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66(4):671–687, 2003.

Juhan Bae, Wu Lin, Jonathan Lorraine, and Roger Grosse. Training data attribution via approximate unrolling. In *Advances in Neural Information Processing Systems*, volume 37, pp. 66647–66686, 2024.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 2397–2430. PMLR, 2023.

Tyler A. Chang, Dheeraj Rajagopal, Tolga Bolukbasi, Lucas Dixon, and Ian Tenney. Scalable influence and fact tracing for large language model pretraining. In *International Conference on Learning Representations*, 2025.

Guillaume Charpiat, Nicolas Girard, Loris Felardos, and Yuliya Tarabalka. Input similarity from the neural network perspective. In *Advances in Neural Information Processing Systems*, volume 32, pp. 5342–5351, 2019.

Sang Keun Choe, Hwijeen Ahn, Juhan Bae, Kewen Zhao, Youngseog Chung, Adithya Pratapa, Willie Neiswanger, Emma Strubell, Teruko Mitamura, Jeff Schneider, Eduard Hovy, Roger Grosse, and Eric P. Xing. What is your data worth to GPT? LLM-scale data valuation with influence functions. In *Advances in Neural Information Processing Systems*, volume 38, pp. 145944–145985, 2025.

Junwei Deng, Ting-Wei Li, Shiyuan Zhang, Shixuan Liu, Yijun Pan, Hao Huang, Xinhe Wang, Pingbang Hu, Xingjian Zhang, and Jiaqi W. Ma. dattri: A library for efficient data attribution. In *Advances in Neural Information Processing Systems*, volume 37, pp. 136763–136781, 2024.

Junwei Deng, Yuzheng Hu, Pingbang Hu, Ting-wei Li, Shixuan Liu, Jiachen T. Wang, Dan Ley, Qirun Dai, Benhao Huang, Jin Huang, Cathy Jiao, Hoang Anh Just, Yijun Pan, Jingyan Shen, Yiwen Tu, Weiyi Wang, Xinhe Wang, Shichang Zhang, Shiyuan Zhang, Ruoxi Jia, Himabindu Lakkaraju, Hao Peng, Weijing Tang, Chenyan Xiong, Jieyu Zhao, Hanghang Tong, Han Zhao, and Jiaqi W. Ma. A survey of data attribution: Methods, applications, and evaluation in the era of generative AI. *SSRN preprint*, 2025. doi: 10.2139/ssrn.5451054. URL `https://ssrn.com/abstract=5451054`.

Junwei Deng, Pingbang Hu, Suliang Jin, Hao Lu, Jiachen T. Wang, Shichang Zhang, and Jiaqi W. Ma. How faithful is trajectory-based data attribution? error sources, remedies, and practical guidelines. *arXiv preprint arXiv:2605.18814*, 2026.

Thomas George, César Laurent, Xavier Bouthillier, Nicolas Ballas, and Pascal Vincent. Fast approximate natural gradient descent in a Kronecker factored eigenbasis. In *Advances in Neural Information Processing Systems*, volume 31, pp. 9550–9560, 2018.

Amirata Ghorbani and James Zou. Data Shapley: Equitable valuation of data for machine learning. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 2242–2251. PMLR, 2019.

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

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

Dirk Groeneveld, Iz Beltagy, Evan Walsh, Akshita Bhagia, Rodney Kinney, Oyvind Tafjord, Ananya Jha, Hamish Ivison, Ian Magnusson, Yizhong Wang, Shane Arora, David Atkinson, Russell Authur, Khyathi Chandu, Arman Cohan, Jennifer Dumas, Yanai Elazar, Yuling Gu, Jack Hessel, Tushar Khot, William Merrill, Jacob Morrison, Niklas Muennighoff, Aakanksha Naik, Crystal Nam, Matthew Peters, Valentina Pyatkin, Abhilasha Ravichander, Dustin Schwenk, Saurabh Shah, William Smith, Emma Strubell, Nishant Subramani, Mitchell Wortsman, Pradeep Dasigi, Nathan Lambert, Kyle Richardson, Luke Zettlemoyer, Jesse Dodge, Kyle Lo, Luca Soldaini, Noah Smith, and Hannaneh Hajishirzi. OLMo: Accelerating the science of language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15789–15809. Association for Computational Linguistics, 2024.

Roger Grosse, Juhan Bae, Cem Anil, Nelson Elhage, Alex Tamkin, Amirhossein Tajdini, Benoit Steiner, Dustin Li, Esin Durmus, Ethan Perez, Evan Hubinger, Kamilė Lukošiūtė, Karina Nguyen, Nicholas Joseph, Sam McCandlish, Jared Kaplan, and Samuel R. Bowman. Studying large language model generalization with influence functions. *arXiv preprint arXiv:2308.03296*, 2023.

Kelvin Guu, Albert Webson, Ellie Pavlick, Lucas Dixon, Ian Tenney, and Tolga Bolukbasi. Simfluence: Modeling the influence of individual training examples by simulating training runs. *arXiv preprint arXiv:2303.08114*, 2023.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

Pingbang Hu, Joseph Melkonian, Weijing Tang, Han Zhao, and Jiaqi W. Ma. GraSS: Scalable data attribution with gradient sparsification and sparse projection. In *Advances in Neural Information Processing Systems*, volume 38, pp. 43880–43904, 2025.

Pingbang Hu, Xueshen Liu, Z. Morley Mao, and Jiaqi W. Ma. Dr. post-training: A data regularization perspective on LLM post-training. *arXiv preprint arXiv:2605.07063*, 2026.

Andrew Ilyas and Logan Engstrom. MAGIC: Near-optimal data attribution for deep learning. *arXiv preprint arXiv:2504.16430*, 2025.

Andrew Ilyas, Sung Min Park, Logan Engstrom, Guillaume Leclerc, and Aleksander Madry. Datamodels: Understanding predictions with data and data with predictions. In *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 9525–9587. PMLR, 2022.

Ruoxi Jia, David Dao, Boxin Wang, Frances Ann Hubis, Nick Hynes, Nezihe Merve Gürel, Bo Li, Ce Zhang, Dawn Song, and Costas J. Spanos. Towards efficient data valuation based on the Shapley value. In *Proceedings of the 22nd International Conference on Artificial Intelligence and Statistics*, volume 89 of *Proceedings of Machine Learning Research*, pp. 1167–1176. PMLR, 2019.

Kevin Fu Jiang, Weixin Liang, James Zou, and Yongchan Kwon. OpenDataVal: A unified benchmark for data valuation. In *Advances in Neural Information Processing Systems (Datasets and Benchmarks Track)*, volume 36, pp. 28624–28647, 2023.

William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.

Daniel M. Kane and Jelani Nelson. Sparser Johnson–Lindenstrauss transforms. *Journal of the ACM*, 61(1):4:1–4:23, 2014.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Pang Wei Koh and Percy Liang. Understanding black-box predictions via influence functions. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1885–1894. PMLR, 2017.

Yongchan Kwon and James Zou. Beta Shapley: A unified and noise-reduced data valuation framework for machine learning. In *Proceedings of the 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pp. 8780–8802. PMLR, 2022.

Yongchan Kwon, Eric Wu, Kevin Wu, and James Zou. DataInf: Efficiently estimating data influence in LoRA-tuned LLMs and diffusion models. In *International Conference on Learning Representations*, volume 2024, pp. 21921–21942, 2024.

Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch distributed: Experiences on accelerating data parallel training. *Proceedings of the VLDB Endowment*, 13(12):3005–3018, 2020. doi: 10.14778/3415478.3415530.

Xuechen Li, Florian Tramèr, Percy Liang, and Tatsunori Hashimoto. Large language models can be strong differentially private learners. In *International Conference on Learning Representations*, 2022.

James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2408–2417. PMLR, 2015.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations*, 2017.

Sung Min Park, Kristian Georgiev, Andrew Ilyas, Guillaume Leclerc, and Aleksander Madry. TRAK: Attributing model behavior at scale. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 27074–27113. PMLR, 2023.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8024–8035, 2019.

Agustin Martin Picard, Lucas Hervier, Thomas Fel, and David Vigouroux. Influenciæ: A library for tracing the influence back to the data-points. In *Explainable Artificial Intelligence (xAI 2024)*, volume 2156 of *Communications in Computer and Information Science*, pp. 193–204. Springer, 2024. doi: 10.1007/978-3-031-63803-9_10.

Garima Pruthi, Frederick Liu, Satyen Kale, and Mukund Sundararajan. Estimating training data influence by tracing gradient descent. In *Advances in Neural Information Processing Systems*, volume 33, pp. 19920–19930, 2020.

Lucia Quirke, Louis Jaburi, David Johnston, William Z. Li, Gonçalo Paulo, Guillaume Martres, Girish Gupta, Stella Biderman, and Nora Belrose. Bergson: An open source library for data attribution. *arXiv preprint arXiv:2606.11660*, 2026.

Andrea Schioppa, Polina Zablotskaia, David Vilar, and Artem Sokolov. Scaling up influence functions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 8179–8186, 2022. doi: 10.1609/aaai.v36i8.20791.

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

TransferLab Team. pyDVL: The Python data valuation library. `https://pydvl.org/stable/`, 2024. Version 0.9.2.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers reinforcement learning. `https://github.com/huggingface/trl`, 2020.

Jiachen T. Wang, Tong Wu, Dawn Song, Prateek Mittal, and Ruoxi Jia. GREATS: Online selection of high-quality data for LLM training in every iteration. In *Advances in Neural Information Processing Systems*, volume 37, pp. 131197–131223, 2024.

Jiachen T. Wang, Prateek Mittal, Dawn Song, and Ruoxi Jia. Data Shapley in one training run. In *International Conference on Learning Representations*, volume 2025, pp. 12358–12395, 2025a.

Jiachen T. Wang, Dawn Song, James Zou, Prateek Mittal, and Ruoxi Jia. Capturing the temporal dependence of training data influence. In *International Conference on Learning Representations*, volume 2025, pp. 11236–11273, 2025b.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45. Association for Computational Linguistics, 2020.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. LESS: Selecting influential data for targeted instruction tuning. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 54104–54132. PMLR, 2024.

Chih-Kuan Yeh, Joon Kim, Ian En-Hsu Yen, and Pradeep Ravikumar. Representer point selection for explaining deep neural networks. In *Advances in Neural Information Processing Systems*, volume 31, pp. 9291–9301, 2018.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. PyTorch FSDP: Experiences on scaling fully sharded data parallel. *Proceedings of the VLDB Endowment*, 16(12): 3848–3860, 2023. doi: 10.14778/3611540.3611569.

A Design Details

This appendix expands the design of §3: the canonical factorization for non-linear layer types (§A.1), the dimension-reduction maps and how they are applied to the factors (§A.2), optimizer-preconditioned capture (§A.3), the routing cost model (§A.4), the capture mechanism (§A.5), and the storage and streaming interfaces that connect capture to attribution (§A.6).

A.1 A canonical factorization for all supported layers

Equation 1 is not specific to linear layers. For every supported layer type, `dattri-LLM` provides an *unfolding map* that reshapes the layer's parameters into a matrix $W \in \mathbb{R}^{N_o \times N_i}$ and expresses its flattened gradient $\gamma = \mathrm{vec}(\mathrm{D}W)$ in the same form, with a_t an unfolded input vector, g_t the corresponding output gradient, and the sum running over T' unfolded positions in place of T. Convolutional layers unfold input patches (`im2col`); transposed convolutions use the same unfolding with the roles of input and output channels exchanged; embedding layers treat the one-hot token id as the input factor, so that materialization scatters g_t into the row of the token; and normalization layers (`LayerNorm`, `RMSNorm`, `GroupNorm`, `InstanceNorm`) use the normalized activation as the input factor, with the bias folded into an extra ones column where present. The supported types are `nn.Linear` (including HuggingFace `Conv1D`), `nn.Conv1d/2d/3d`, `nn.ConvTranspose1d/2d/3d`, `nn.Embedding` and `nn.EmbeddingBag`, and the normalization modules above. Because every operation in the library depends on a layer only through Eq. 1, a single implementation of each operation, and hence of each attribution method, covers all of them. Trainable parameters outside these families can still be captured through a batch-level parameter-gradient hook, which yields the materialized form directly.

A.2 Dimension reduction

§3.2 introduces factor projection (Eq. 2) and materialized projection. This section records how each is applied and which random maps are available.

Factor projection. The projection matrices P_a and P_g are applied inside the backward hook, so the raw factors are never buffered and only the projected pair $(\tilde{a}_t, \tilde{g}_t)$ is retained. Because the result has the form of Eq. 1 with (N_i, N_o) replaced by (k_a, k_g), every operation defined on factorized gradients, including routing (§A.4) and the Kronecker-factored statistics, applies to the projected factors without change. When $k_a = k_g$ the construction coincides with the convention of LoGRA (Choe et al., 2025) and GraSS (Hu et al., 2025). The two sides use distinct seeds derived from a per-layer base seed, which must be held fixed across every gradient that will be compared, including gradients captured at different steps or on different ranks.

Materialized projection. A single matrix $P \in \mathbb{R}^{k_{\text{dense}} \times (N_i N_o)}$ is applied to the flattened gradient, $\tilde{\gamma} = P\gamma \in \mathbb{R}^{k_{\text{dense}}}$, as in TRAK (Park et al., 2023). For normalization layers, materialized projection is the only projection available. Their factors are captured like those of any other layer, but their weight gradient is an elementwise product of the two factors summed over positions, $\sum_t \hat{x}_t \odot g_t$, rather than an outer product, so factor projection does not apply; the dense gradient, with one entry per feature, is also smaller than the factors it is computed from. Materialized projection is incompatible with the Kronecker-factored methods, which require the factors.

Random maps. Either projection may use a dense Gaussian map (Johnson & Lindenstrauss, 1984), a Rademacher map with entries $\pm 1/\sqrt{d_{\text{out}}}$, where d_{out} is the map's output dimension (k_a, k_g, or k_{dense}) (Achlioptas, 2003), a sparse Johnson–Lindenstrauss transform with a fixed number of nonzeros per column (Kane & Nelson, 2014), or the sparsify-then-project map of GraSS (Hu et al., 2025). In addition, a *random mask* keeps a fixed random subset $\mathcal{M}$ of the entries of $\mathrm{D}W$. Because entry (r, c) of $\mathrm{D}W$ equals $\sum_{t=1}^{T} g_{t,r}\, a_{t,c}$, the masked entries are gathered directly from the factors as a sum of $|\mathcal{M}|$ products per token position, without forming $\mathrm{D}W$. The mask is the one reduction that preserves the coordinate identity of each retained entry, which is what optimizer preconditioning (§A.3) requires. All maps are generated from the same per-layer seed so that projected gradients from different steps, ranks, and passes remain comparable.

A.3 OPTIMIZER-PRECONDITIONED CAPTURE

Modern models are trained with AdamW rather than SGD, and a growing family of attribution methods scores the *update* a training example induces rather than its raw gradient: LESS (Xia et al., 2024) compares the Adam direction of an example with a query gradient, and AdamW-influence (Deng et al., 2026) unrolls that direction through the remaining AdamW steps. For Adam-family optimizers, Eq. 3 maps sample i's flattened gradient $\gamma^{(i)}$ to its update direction $\Gamma^{(i)}$ at optimizer step τ, using the moment estimates (m, v) before the coming `optimizer.step()`. Eq. 3 has properties that shape the design. It needs the optimizer state *at the step the example was used*, so it must be evaluated during training; and it is nonlinear in $\gamma^{(i)}$ coordinate by coordinate, so it does not commute with a random projection, which mixes coordinates.

`dattri-LLM` applies optimizer preconditioning to gradient entries recovered from the factors. `HookManager` accepts the training optimizer, and for each hooked layer the per-example gradient entries are recovered from the factors and mapped through the optimizer's update rule before they are buffered, so only the preconditioned copy ever exists, at the same place and with the same cost profile as factor projection. The random mask of §A.2 pairs naturally with it: the mask selects the coordinates, the entries are recovered from the factors, and the rule is applied to exactly those entries. Every coordinate-wise optimizer in `torch.optim` is supported through one kernel, and the map is switchable per pass so that a raw query pass can share the hooks of a preconditioned training pass. For trajectory methods a callback additionally records the moments on the same coordinates before and after every update, from which the batch gradient the optimizer consumed is recovered exactly. On this capture path, LESS is TracIn over preconditioned gradients with a cosine metric, in either its per-step or its checkpoint form, and AdamW-influence is a backward recurrence over the stored steps. The recurrence is linear in its summary matrix, so it can be carried either on the training side, at $O(|\mathcal{M}|^2)$ memory in the mask size $|\mathcal{M}|$, or on the query side through the query gradients, which removes the mask entirely at a cost linear in the number of queries (§4.2).

A.4 FLOP-AWARE ROUTING

Consider the cross-Gram matrix of gradient inner products between two batches of sizes B_1 and B_2, with T tokens per example and layer dimensions N_i, N_o. Table 2 summarizes the factorized and materialized routes.

Table 2: Complexity of computing a cross-Gram matrix between two batches of per-example gradients. Space denotes the storage required for the gradient representations and excludes the $O(B_1B_2)$ output matrix common to both routes. Under factor projection (Eq. 2), N_i and N_o are replaced by the projection dimensions k_a and k_g.

Route	Arithmetic operations	Representation space
Factorized	$O\big(B_1B_2T^2(N_i + N_o)\big)$	$O((B_1 + B_2)T(N_i + N_o))$
Materialized	$O((B_1 + B_2)TN_iN_o + B_1B_2N_iN_o)$	$O((B_1 + B_2)N_iN_o)$

Regimes. The materialized route pays a one-time construction cost of TN_iN_o operations per example, amortized across all B_1B_2 pairwise comparisons of cost N_iN_o each; the factorized route avoids construction but pays $T^2(N_i + N_o)$ per pair. Short sequences and wide layers therefore favor the factorized route, and long sequences or large comparison batches favor materialization. The storage comparison has a closed form: the factorized form is smaller exactly when $T < N_iN_o/(N_i+N_o)$. For unprojected wide layers this threshold is on the order of the layer width, so factors are kept; under factor projection it shrinks to $k_ak_g/(k_a + k_g)$, which realistic sequence lengths far exceed, so projected factors are better materialized as small $k_g \times k_a$ gradient matrices. The projected and unprojected regimes thus select opposite representations under the same rule.

Routing. Routing is evaluated per layer and per operation, for cross-Gram matrices and for per-example norms alike, by comparing the operation counts above and taking the cheaper route; the same routed kernels serve every attribution method and the data-selection callback. At capture time the representation that is persisted follows the storage rule and is exposed as a per-layer setting. Because the routes are algebraically equivalent, routing does not change attribution scores in exact

arithmetic. Appendix B.4 analyzes the case of reused query representations and evaluates routing across sequence lengths.

A.5 CAPTURE MECHANISM

Hook families. `HookManager` attaches a hook family to each selected layer. The default *factorized* family registers a forward hook that records the layer input and a full backward hook that records the output gradient. A *parameter-gradient* family registers a tensor hook on each parameter and records the batch-level gradient in materialized form; it is also the fallback for trainable layers outside the supported types. Layers are selected by type, by regular expression, or by an all-eligible marker, and layers whose type is not recognized automatically, such as custom normalization modules, are declared through per-layer overrides rather than by editing the model. When the training call is buried inside a framework, the capture context can be opened and closed from a framework callback instead of a `with` block.

Loop-independent step detection. Because the manager never sees the training loop, it infers when a step has finished from the hooks themselves. It tracks which hooked layers participated in the current forward pass and waits until every one has fired its backward hook, with an additional end-of-backward barrier for parameter gradients; layers skipped by the current control path are not waited for. Repeated invocations of a layer within a step (weight tying, multiple forward calls) are tracked as separate virtual layers. Under gradient checkpointing, the recomputed forward pass is recognized by the presence of a live autograd task and matched to the correct backward pass rather than double-counted. For backward passes that must not contribute, such as a validation pass used to form a target gradient, the manager can save, clear, and restore its per-step state around the extra pass, so the training step is captured exactly as if the extra pass had not occurred.

Parallelism. Under DDP (Li et al., 2020), parameter hooks fire when a parameter's local gradient is computed, before the all-reduce, so each rank captures the gradient of its own batch; under FSDP (Zhao et al., 2023), per-parameter gradient hooks do not fire, so step completion is signaled through an end-of-backward callback queued on the autograd engine. In both cases each rank hooks its local shard and writes records independently, and the storage manager merges the per-rank indexes at read time, so sharded capture reproduces single-device gradients. The data-selection callback's edit of `param.grad` is collective: contributions are all-reduced and applied to the full tensor under DDP or to the owned shard under FSDP, and under DDP the edit is queued to run after the reducer's own write-back.

Integration with training frameworks. Figure 6 shows the capture code for the HuggingFace `Trainer`, TRL's `GRPOTrainer`, and the OLMo trainer. In each case the trainer is constructed and called exactly as without attribution: the user builds a `HookManager` over the model, attaches callbacks that offload the captured gradients, and runs the unmodified training call inside `hm.collect()`. The optional layer selectors are provided and can be tailored for different model architectures.

A.6 STORAGE, IDENTITY, AND THE CAPTURE–ATTRIBUTION INTERFACE

Sample identity. Offloaded gradients are keyed by a content hash of a sample's model inputs, which is independent of position and shuffling. The on-disk index maps `input_hash` $\rightarrow$ `(step, sample_idx)` $\rightarrow$ file, so retrieving one sample's gradient is a direct slice rather than a scan of the batch record it was written in.

Store. Records are written before they are indexed and the index is persisted atomically, so an interrupted run leaves the store readable up to the last indexed record rather than corrupt. Under distributed training, each rank writes to its own subdirectory and a freshly constructed manager merges the per-rank indexes, so the number of ranks used for capture is invisible to the reader. The offload callback can merge the micro-batches of a gradient-accumulation window into one record stamped with the optimizer step, which is exact because parameters do not change within a window.

(a) HuggingFace `Trainer`

```python
from dattri_llm import (GradientStorageManager, HookManager,
                        HookManagerConfig, OffloadCallback)
store = GradientStorageManager("gradients/")
hm = HookManager(model,
    config=HookManagerConfig(linear_io=[r"transformer\.h\.", r"wte", r"lm_head"]),
    callbacks=[OffloadCallback(offload_interval=1, file_manager=store)])
# Pattern A: wrap the unmodified training call
trainer = Trainer(model=model, args=training_args, train_dataset=dataset)
with hm.collect():
    trainer.train()
# Pattern B: open and close capture from a Trainer callback
class CollectCallback(TrainerCallback):
    def __init__(self, hm):
        self.hm, self.stack = hm, contextlib.ExitStack()
    def on_train_begin(self, args, state, control, **kwargs):
        self.stack.enter_context(self.hm.collect())
    def on_train_end(self, args, state, control, **kwargs):
        self.stack.close()
trainer = Trainer(model=model, args=training_args, train_dataset=dataset,
                  callbacks=[CollectCallback(hm)])
trainer.train()
```

(b) TRL `GRPOTrainer`

```python
trainer = GRPOTrainer(model=model, reward_funcs=reward_fn, args=grpo_config,
                      train_dataset=prompts, processing_class=tokenizer)
hm = HookManager(model,
    config=HookManagerConfig(linear_io=[r"transformer\.h\."]),
    callbacks=[OffloadCallback(offload_interval=1, file_manager=store)])
with hm.collect():
    trainer.train()
```

(c) OLMo `Trainer`

```python
trainer = Trainer(cfg=cfg, model=model, dist_model=dist_model, optim=optim,
                  scheduler=scheduler, train_loader=loader, device=device,
                  evaluators=[])
hm = HookManager(model,
    config=HookManagerConfig(linear_io=[r"ff_proj", r"ff_out"]),
    callbacks=[OffloadCallback(offload_interval=1, file_manager=store)])
with hm.collect():
    trainer.fit()
```

Figure 6: Gradient capture around three training frameworks. The trainer and its training call are unmodified.

Streaming interface. Attributors consume gradients through one contract: a source yielding per-step `(step, gradient, hashes)` blocks. A disk-backed source re-iterates over pre-collected gradients, loading one file per item so a standard data loader can prefetch whole blocks. A live source computes blocks from a forward and backward pass, either as a re-iterable probe at a fixed checkpoint or, in its updating mode, advancing the optimizer so each step is a genuine point on a training trajectory; the updating mode mirrors the HuggingFace `Trainer`'s inner loop (optimizer and scheduler construction, mixed precision, clipping, and distributed wrapping). Attributors branch on exactly one property of the source, whether it is re-iterable, which methods with a pre-pass such as K-FAC's Fisher fit require. Scores are returned as an `AttributionScore` whose rows are (example, step) pairs, so trajectory-aware and trajectory-agnostic methods share one container.

Workflows. The same attributor code serves the *on-the-fly* workflow, where an attributor drives a live source over the model and data and scores in one run, and the *store-then-attribute* workflow, where an ordinary training run caches gradients via the offload callback and attribution is performed later over the store, without the model and without another backward pass, so different methods and settings can be re-run over the same cache.

Table 3: Qwen models used in the scaling experiments: parameters (B), transformer blocks L, model width d, MLP width d_{ff}, attention heads (query / key-value), and hooked linear layers ($7L$). The Qwen family provides a smooth progression in model size from 0.5B to 110B.

Scale	Checkpoint	Params	L	d	d_{ff}	Heads	Hooked
0.5B	Qwen2.5-0.5B	0.49	24	896	4864	14/2	168
1.5B	Qwen2.5-1.5B	1.54	28	1536	8960	12/2	196
3B	Qwen2.5-3B	3.09	36	2048	11008	16/2	252
7B	Qwen2.5-7B	7.62	28	3584	18944	28/4	196
14B	Qwen2.5-14B	14.8	48	5120	13824	40/8	336
32B	Qwen2.5-32B	32.5	64	5120	27648	40/8	448
72B	Qwen2.5-72B	72.7	80	8192	29568	64/8	560
110B	Qwen1.5-110B	111	80	8192	49152	64/8	560

B Experiment Details

B.1 Throughput Under a Fixed Hardware Budget

We evaluate the Qwen family of models in Table 3 on WikiText-103 with sequence length 512 and bf16 precision. Each configuration scores 512 training sequences against one query through 32B and four queries at 72B and 110B. The query count is matched across libraries at each scale. Each run receives four H200 GPUs (141 GB per device), 32 CPU cores, and 256 GB of host memory. Attention and MLP linear layers are included in attribution; embeddings and the language-model head are excluded.

Library configurations. `dattri-LLM`, Bergson and Kronfluence use FSDP. LogIX uses one model replica per device under DDP. Factor projection with $k_a = k_g = 64$ is requested where supported. Kronfluence runs without that projection. Consequently, the throughput comparison reflects the capabilities of the evaluated implementations as well as their execution efficiency. For each library, method, and model size, we manually test power-of-two per-device batch sizes and select the largest that can successfully complete the workload.

Timing. After two untimed warm-up steps, the GPU-synchronized timed interval covers the complete attribution computation: gradient capture, Kronecker-factor estimation and eigendecomposition where required, and scoring. Model loading, data preparation, and process initialization are excluded. For Bergson, which launches workers for successive pipeline stages, the reported time further removes the measured worker initialization and model-loading intervals for fairness. Throughput is 512 divided by the resulting elapsed time; it measures the complete finite workload and includes necessary per-call overheads for each attribution method.

Memory measurements. Table 5 reports peak per-device memory at the selected batch size. For `dattri-LLM`, LogIX, and Kronfluence we use `torch.cuda.max_memory_allocated`; Bergson uses NVML readings because its computation runs in child processes. The memory comparison at a common batch size is further reported below.

Memory Scaling at Batch Size One We evaluate GPU memory usage at batch size one to assess the largest model each library can accommodate, using the same models and workload as above. Each configuration is evaluated on one H200; when a single device is insufficient, we use four H200s if the library supports sharding. LogIX does not support sharding in its current implementation. Table 6 explicitly marks completed four-device sharded runs and reports their estimated aggregate memory usage across all four devices. A dash indicates that the configuration runs out of memory under this evaluation protocol.

B.2 Fidelity Evaluation Details

We evaluate three models from three families: GPT-2 (124M), Qwen2.5-1.5B, and OLMo-3-7B (Groeneveld et al., 2024). For each model, all parameters are trained for one epoch on 512

Table 4: Throughput (training sequences/s) on four H200s, with the per-device batch size in parentheses. The workload contains 512 training sequences and one query through 32B, and four queries at 72B and 110B. A dash denotes a configuration reported as out of memory.

Method	Library	0.5B	1.5B	3B	7B	14B	32B	72B	110B
GRADDOT	`dattri-LLM`	613 (128)	299 (64)	186 (32)	110 (32)	49.9 (16)	22.0 (8)	8.6 (8)	6.9 (4)
	Bergson	68.8 (64)	57.9 (64)	43.2 (32)	39.8 (32)	24.6 (16)	14.3 (8)	6.1 (8)	4.9 (4)
	Kronfluence	177 (64)	111 (32)	64.4 (16)	41.5 (16)	—	—	—	—
	LogIX	65.8 (128)	46.5 (64)	34.8 (32)	34.7 (32)	17.2 (16)	10.4 (8)	—	—
K-FAC	`dattri-LLM`	375 (128)	174 (64)	120 (32)	92.2 (32)	40.3 (16)	16.8 (8)	7.4 (8)	5.9 (4)
	Bergson	2.8 (32)	1.9 (32)	1.2 (16)	0.8 (8)	—	—	—	—
	Kronfluence	9.9 (64)	2.8 (32)	1.3 (16)	—	—	—	—	—
	LogIX	61 (128)	44.8 (64)	31.5 (32)	32.6 (32)	16.4 (16)	9.7 (8)	—	—
EK-FAC	`dattri-LLM`	337 (128)	174 (64)	116 (32)	83.9 (32)	38.6 (16)	16.1 (8)	7.2 (8)	5.6 (4)
	Bergson	2.9 (32)	1.7 (16)	1 (8)	0.5 (4)	—	—	—	—
	Kronfluence	8.3 (64)	2.3 (32)	1 (8)	—	—	—	—	—
	LogIX	49.9 (128)	37.5 (64)	25.6 (32)	24.2 (32)	12.2 (16)	6.9 (8)	—	—

Table 5: Peak per-device GPU memory (GB) at the batch sizes used in Table 4. Batch sizes vary across methods, libraries, and scales; these values document the throughput configurations and do not constitute a fixed-batch memory comparison.

Method	Library	0.5B	1.5B	3B	7B	14B	32B	72B	110B
GRADDOT	`dattri-LLM`	117.6	93.0	67.2	88.4	72.0	79.0	131.0	127.8
	Bergson	80.9	119.0	86.5	104.6	87.7	91.2	139.8	135.0
	Kronfluence	101.7	95.3	80.7	119.7	—	—	—	—
	LogIX	117.4	94.6	70.9	97.4	91.0	124.4	—	—
K-FAC	`dattri-LLM`	117.6	93.0	67.2	88.4	72.0	80.0	132.1	127.8
	Bergson	95.4	134.4	112.4	132.5	—	—	—	—
	Kronfluence	101.7	95.3	98.6	—	—	—	—	—
	LogIX	117.4	94.6	70.9	96.8	91.0	124.4	—	—
EK-FAC	`dattri-LLM`	117.6	93.0	67.2	88.4	72.0	80.0	132.1	127.8
	Bergson	132.5	121.2	109.2	134.1	—	—	—	—
	Kronfluence	108.7	122.6	100.4	—	—	—	—	—
	LogIX	117.4	94.3	71.4	97.4	91.0	124.4	—	—

blocks of 128 WikiText-2 tokens, drawn with the model's own tokenizer, using batch size 32 for 16 AdamW updates. The optimizer uses $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$, and zero weight decay. The learning rate peaks at 10^{-5} after linear warm-up over the first 10% of steps, followed by linear decay. GPT-2 and Qwen2.5 are trained by the evaluation's own training loop. The OLMo models are trained by OLMo-core, the trainer of the OLMo releases, without modification: the released weights are converted once into an OLMo-core checkpoint, the learning-rate schedule is passed as a scheduler configuration, and `dattri-LLM` captures gradients by wrapping the trainer's `fit()` call, on one GPU. For TSLOO (Wang et al., 2025b), each of 50 randomly selected training blocks is removed from its original batch in a separate replay of training. We record the change in loss on 64 held-out validation blocks. Fidelity is the Spearman correlation between attribution scores and these loss changes over the 50 training blocks, computed separately for each validation block and averaged over the 64 blocks. For numerical precision, we run training and evaluation using fp32, with TF32 disabled, eager attention, deterministic kernels, and dropout disabled. Pretrained weights are loaded once, and the reference run and all 50 removal runs start from the same copy. Each model scale uses a fixed seed, and we average over the 64 query correlations to get robust results.

Methods. `dattri-LLM` implements AdamW-influence with full-dimensional gradients and with random coordinate masks. The reduced configuration uses ten disjoint masks of 512 or 8192 coordinates per layer, computes scores independently for each mask, and averages them. Its measured

Table 6: GPU memory usage (GB) at batch size one, with 64 training sequences and one query. ^{F} denotes a completed sharded run on four H200s, with memory reported as an estimated aggregate across all four devices. All other completed runs use one H200. A dash indicates an out-of-memory failure under the evaluation protocol.

Method	Scale	`dattri-LLM`	Bergson	LogIX	Kronfluence
GradDot	0.5B	2	2.1	2	3.7
	1.5B	4.8	4.8	4.9	10.2
	3B	8.7	8.5	8.6	19.9
	7B	18.8	17.3	17.8	43.5
	14B	33.8	32.2	33.2	85
	32B	72.1	69.9	69.8	—
	72B	323^{F}	258^{F}	—	—
	110B	476^{F}	354^{F}	—	—
K-FAC	0.5B	2.4	12	2	8.9
	1.5B	6.7	45	4.9	31.7
	3B	12.9	90.8	8.2	61.8
	7B	28.5	440^{F}	17.8	—
	14B	55.4	—	33.2	—
	32B	122.2	—	71.9	—
	72B	324^{F}	—	—	—
	110B	476^{F}	—	—	—
EK-FAC	0.5B	2.4	12.7	2.1	10.8
	1.5B	6.7	45.1	4.9	37.6
	3B	12.9	90.8	8.2	73.7
	7B	28.5	440^{F}	17.8	—
	14B	55.4	—	33.2	—
	32B	122.2	—	69.8	—
	72B	324^{F}	—	—	—
	110B	476^{F}	—	—	—

cost includes all ten masks. We also evaluate full-dimensional EK-FAC and EK-FAC with factor projection $k_a = k_g = 64$. From Bergson (v0.26.1), we evaluate MAGIC (Ilyas & Engstrom, 2025), SOURCE (Bae et al., 2024), TrackStar (Chang et al., 2025), and EK-FAC (George et al., 2018). All EK-FAC implementations use damping equal to 0.1 times the mean eigenvalue of each layer. TrackStar uses per-module projection dimension 64. SOURCE uses four segments of four steps with two checkpoints per segment. MAGIC uses Bergson's functional trainer to replay the trajectory and differentiate each query loss through the training run; its replay agrees with the reference within 10^{-5} in validation loss. On OLMo-3-7B the Bergson method that scores a trained model (TrackStar) takes it through its HuggingFace export; MAGIC and SOURCE own the training loop and cannot run under OLMo-core's trainer. Every method scores the same 64 queries against the same training trajectory.

Batch sizes. Every method captures the training gradients in batches of 8 blocks. The full-dimension methods score the 64 queries in chunks, and the chunk is, for each library, the largest power of two that completes within the budget: 64, 8, and 4 queries for `dattri-LLM`'s EK-FAC on GPT-2, Qwen2.5-1.5B, and OLMo-3-7B, and 16, 1, and N/A for Bergson's EK-FAC and SOURCE. At 7B, `dattri-LLM`'s EK-FAC keeps its Kronecker factors in host memory and moves one layer's factors to the GPU at a time.

Resource budget. Each execution receives one B200 (183 GB), 16 CPU cores, 256 GB of host memory, and 1 TB of local disk; the 7B model's fp32 training state (16 bytes per parameter for weights, gradients, and the two Adam moments) does not fit a smaller GPU. The reported time is the attribution of all 64 queries: training the trajectory, loading the model and the data, and the TSLOO reference construction are excluded, since a model is trained whether or not it is attributed. For the single-checkpoint methods the time runs from the trained model to the scores; for the trajectory-based methods it covers the attribution pass over the recorded trajectory, which for MAGIC is the

Table 7: Resource limits in the fidelity benchmark under the budget of one B200, 256 GB of host memory, and 1 TB of local disk.

Method	Model	Limiting resource
MAGIC (Bergson)	Qwen2.5-1.5B	GPU memory
AdamW-influence (full, `dattri-LLM`)	Qwen2.5-1.5B	GPU memory
MAGIC	OLMo-3-7B	GPU memory
AdamW-influence (full)	OLMo-3-7B	GPU memory
SOURCE (Bergson)	OLMo-3-7B	Local disk
EK-FAC (Bergson)	OLMo-3-7B	GPU memory

Table 8: Complete fidelity results on one B200: Spearman correlation with TSLOO (ρ), attribution time of the 64 queries in seconds (t_{attr}), and peak GPU memory in GB. "–": infeasible under the budget.

		GPT-2			Qwen2.5-1.5B			OLMo-3-7B		
Library	Method	ρ	t_{attr}	GB	ρ	t_{attr}	GB	ρ	t_{attr}	GB
`dattri-LLM`	AdamW-inf. (k=512)	.680	10	8	.526	51	47	.437	73	136
`dattri-LLM`	AdamW-inf. (k=8192)	.762	28	8	.639	103	49	.633	170	138
`dattri-LLM`	AdamW-inf. (full)	.845	1137	25	–	–	–	–	–	–
`dattri-LLM`	EK-FAC (k_a=k_g=64)	.699	15	8	.420	50	47	.303	126	40
`dattri-LLM`	EK-FAC (full)	.664	17	30	.434	447	96	.402	3444	139
Bergson	MAGIC	.833	1265	24	–	–	–	–	–	–
Bergson	SOURCE	.773	846	13	.275	28026	64	–	–	–
Bergson	TrackStar	.523	34	8	.332	289	47	.374	368	56
Bergson	EK-FAC	.681	171	13	.432	4560	64	–	–	–

backward pass through training. Throughput in Figure 3 is 64 queries divided by this time. Table 7 further records the limiting resource.

B.3 Fixed-Batch Comparisons at a Fixed Model Size

In addition to the maximum-throughput analysis in the main text, we further evaluate all libraries at the same batch size to examine performance differences under matched conditions. We compare `dattri-LLM`, LogIX, Bergson, and Kronfluence on Pythia-410M (Biderman et al., 2023), attributing language-modeling loss on 1024 WikiText-103 training sequences of length 512, with either one or 16 queries. Completed configurations use training batch size 8 on one A40 (46 GB) in fp32. We evaluate both factor projection with $k_a = k_g = 64$ and full-dimensional gradients; projection uses a capture-time Rademacher map where configurable. Table 9 reports runtime and memory usage for both query counts. Note that these supplementary experiments use a different model and hardware configuration from the Qwen throughput benchmark.

Results. `dattri-LLM` has the lowest runtime in nine of the twelve rows of Table 9, and in every row that uses a curvature-based method. For projected K-FAC and EK-FAC it runs 1.6–2.5× faster than LogIX and 3.3–4.1× faster than Bergson; for the full-dimensional variants it runs 1.4–1.6× faster than Kronfluence and 1.2–2.1× faster than Bergson. The remaining three rows are GradDot, where the runtimes of `dattri-LLM`, Bergson, and Kronfluence differ by at most two seconds. At this scale a single forward and backward pass over the 1024 training sequences in fp32 accounts for most of the runtime, and GradDot adds little to it, so any library that runs the model once lands near the same floor; LogIX, which extracts gradients one query at a time through a disk-backed store, does not. The curvature-based methods add a fit over the training gradients and a per-layer preconditioning of the queries, and it is this attribution-side work that separates the libraries. Moving from one query to 16 leaves the runtime of `dattri-LLM` almost unchanged, whereas Bergson's full-dimensional K-FAC and EK-FAC take 1.6–1.7× longer.

Table 9: **Cross-library efficiency under matched batch sizes.** Results use Pythia-410M on one A40 (46 GB), with 1024 WikiText-103 training sequences and either one or 16 queries. t_{attr}: attribution wall-clock time (s), excluding model construction and data loading; M_{GPU}: peak GPU memory (GB) over the same phases. The lowest runtime in each row is bold. Completed configurations use training batch size 8. OOM indicates an out-of-memory failure at every tested batch size down to one; N/A indicates an unsupported combination.

		dattri-LLM		LogIX		Bergson		Kronfluence	
Method	Proj.	t_{attr}	M_{GPU}	t_{attr}	M_{GPU}	t_{attr}	M_{GPU}	t_{attr}	M_{GPU}
One query ($n_{\text{test}} = 1$)									
GradDot	$k_a{=}k_g{=}64$	41	7	68	7	**39**	8	N/A	N/A
	full	**45**	11	OOM	—	53	13	50	11
K-FAC	$k_a{=}k_g{=}64$	**54**	10	89	7	180	20	N/A	N/A
	full	**134**	19	OOM	—	194	20	201	12
EK-FAC	$k_a{=}k_g{=}64$	**58**	10	143	7	N/A	N/A	N/A	N/A
	full	**223**	21	OOM	—	278	27	347	15
16 queries ($n_{\text{test}} = 16$)									
GradDot	$k_a{=}k_g{=}64$	40	13	71	7	**39**	8	N/A	N/A
	full	58	31	OOM	—	121	30	**57**	28
K-FAC	$k_a{=}k_g{=}64$	**55**	18	90	7	225	44	N/A	N/A
	full	**153**	34	OOM	—	317	44	210	28
EK-FAC	$k_a{=}k_g{=}64$	**59**	18	146	7	N/A	N/A	N/A	N/A
	full	**230**	38	OOM	—	461	44	360	28

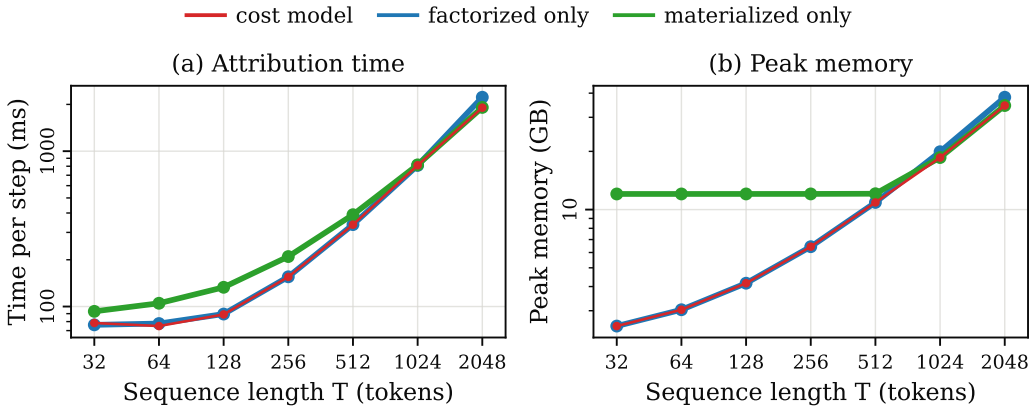

Figure 7: **Adaptive routing versus fixed gradient representations.** Mean attribution time per step (left) and peak GPU memory (right) for full-dimensional GradDot on Pythia-410M as sequence length increases, using batch size 8, one query, and fp32 on one A40. The cost-model heuristic selects between factorized and materialized execution; fixed-route ablations use the same implementation. Adaptive routing closely tracks the faster fixed configuration while maintaining the lowest or tied peak memory at every evaluated length. Memory includes all GPU allocations.

B.4 Routing Across Sequence Lengths

In this section, we show the effectiveness of the heuristic proposed in §3.2. We use the full-dimensional GradDot setting of Appendix B.3: Pythia-410M, WikiText-103, fp32, one query, and one A40. Sequence length ranges from 32 to 2048 tokens, and all configurations use batch size 8. We time 128 steps after 8 warm-up steps and report mean attribution time per step and peak GPU memory. The three `dattri-LLM` configurations differ only in their routing decision; the fixed-route runs are ablations of the same implementation. Figure 7 presents the curves.

Table 10: Speedup of weight-gradient-free capture over standard capture (standard time / weight-gradient-free time) for `dattri-LLM` on Pythia-410M, one L40S (48 GB), fp32, with the workload of Table 9. Each entry is the median over five paired repetitions.

Method	Projection	$n_{\text{test}} = 1$	$n_{\text{test}} = 16$
GRADDOT	$k_a = k_g = 64$	1.19×	1.18×
GRADDOT	full	1.16×	1.09×
K-FAC	$k_a = k_g = 64$	1.14×	1.11×
K-FAC	full	1.10×	1.07×
EK-FAC	$k_a = k_g = 64$	1.10×	1.14×
EK-FAC	full	1.08×	1.08×

Adaptive versus fixed execution. The maximum runtime excess over the faster fixed configuration is approximately 2.8%. For $32 \le T \le 512$, materialized execution takes 16–50% longer than adaptive routing. At $T = 2048$, factorized execution takes approximately 17% longer. Noticeably, while the heuristic is developed for runtime optimization, the short-sequence memory advantage is also substantial: at $T = 32$, the always-materialized run has 4.8× the peak memory of adaptive execution. These are total GPU-memory measurements, not isolated gradient-buffer sizes. Adaptive routing has the lowest or tied memory among the three `dattri-LLM` configurations at every evaluated length.

B.5 TOKEN-LEVEL ATTRIBUTION EXAMPLES

We illustrate token-level attribution using GRADDOT, K-FAC, and EK-FAC on Pythia-410M and WikiText-103. For these methods, the sequence-level score decomposes over the training sequence's token positions. For a single layer, let $R^{(j)} \in \mathbb{R}^{N_o \times N_i}$ denote the query-side matrix for query j. The contribution of position t in training sample i is

$$s_t^{(i,j)} = (g_t^{(i)})^\top R^{(j)} a_t^{(i)}, \qquad \sum_{t=1}^{T_i} s_t^{(i,j)} = \left\langle \mathrm{D}W^{(i)}, R^{(j)} \right\rangle_F .$$

Here $R^{(j)} = \mathrm{D}W^{(j)}$ for GRADDOT; for K-FAC and EK-FAC, it is the corresponding preconditioned query gradient reshaped into a matrix. Contributions are summed across attributed layers to obtain the reported token scores, whose sum recovers the sequence-level score. All attention and MLP linear layers are attributed without projection. K-FAC and EK-FAC fit curvature on 32 WikiText-103 paragraphs with damping 10^{-3}. Each panel is normalized independently because the methods produce scores on different scales. The figures illustrate reuse of the captured factors for fine-grained analysis; they do not establish a token decomposition for every attribution method in the library.

B.6 WEIGHT-GRADIENT-FREE CAPTURE FOR FROZEN ATTRIBUTION

Attribution at a fixed checkpoint reads only the inputs and output gradients of the hooked layers; the weight gradients that the backward pass also computes are never used. `dattri-LLM` therefore provides an optional capture mode (`invasive_linear_io`) that, during capture, routes each hooked `nn.Linear` through an operation whose backward pass returns only the input gradient. It enables more efficient exact attribution by eliminating the memory required for parameter gradients and reducing the runtime overhead of gradient collection. Because the mode produces no parameter gradients, it is applicable only outside training; capture within a training loop, including online data selection, uses standard capture. Kronfluence and Bergson avoid the same computation by freezing the model parameters during attribution, whereas LogIX has to compute weight gradients for the layers it tracks. In Table 9, `dattri-LLM` uses this mode for GRADDOT and full-dimensional K-FAC and EK-FAC, and standard capture for projected K-FAC and EK-FAC. To quantify its effect, we rerun each `dattri-LLM` configuration of Table 9 under both modes, changing only the capture mode, on one L40S (48 GB) in fp32 with five paired repetitions. As Table 10 shows, standard capture takes 7–19% longer and uses at most 4.5 GB more peak memory, which is the cost of the weight-gradient computation it retains.

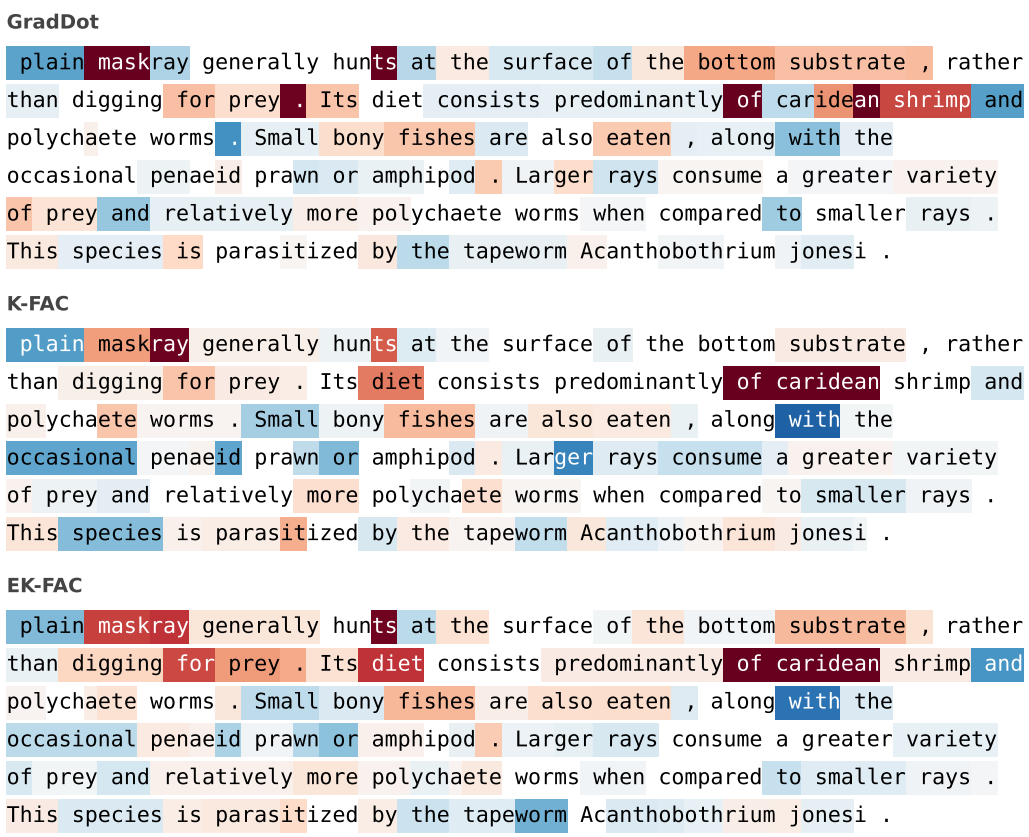

Figure 8: **maskray** (train row 308). Target: "The plain maskray is a species of stingray that feeds on caridean → **shrimp**". Under GRADDOT the largest positions are the subword `ts` of *hunts*, *mask*, and the two occurrences of *of*, with *shrimp* itself at rank 6. Both preconditioned methods concentrate on *of caridean*, the phrase the target completes.

B.7 DATA SELECTION DETAILS

Training uses Llama-3.2-1B with AdamW, a linear learning-rate schedule with 3% warm-up, gradient clipping at 1.0, and batch size 8, over three seeds. The learning rate is 10^{-6} for full-parameter fine-tuning and 10^{-4} for LoRA. Master weights are stored in fp32, with bf16 autocast during training. The source pool is Alpaca and the target task is SAMSum, with the same chat template applied to both datasets. At each step, the scoring target is one example from a 16-example SAMSum validation pool. Selection removes the bottom half of each batch according to GRADDOT scores against that example. Full-parameter runs attribute all transformer attention and MLP linear layers, excluding embeddings and the language-model head. LoRA uses rank 8 and $\alpha = 16$ on the same projections; capture and gradient intervention are restricted to the adapter layers. Test perplexity is evaluated every 50 steps on the response tokens of 500 held-out SAMSum examples, with prompt tokens masked.

Runtime. Figure 11 reports per-step runtime under full-parameter fine-tuning, averaged over steps after warm-up. Standard training takes 89 ms per step, selection with `dattri-LLM` takes 146 ms, and the dedicated implementation of Hu et al. (2026) takes 154 ms. The library implementation is approximately 5% faster than the dedicated implementation. The dedicated implementation uses an additional forward pass to obtain training gradients, whereas `dattri-LLM` captures the required quantities from the training backward pass. Scoring and selection account for the remaining application overhead.

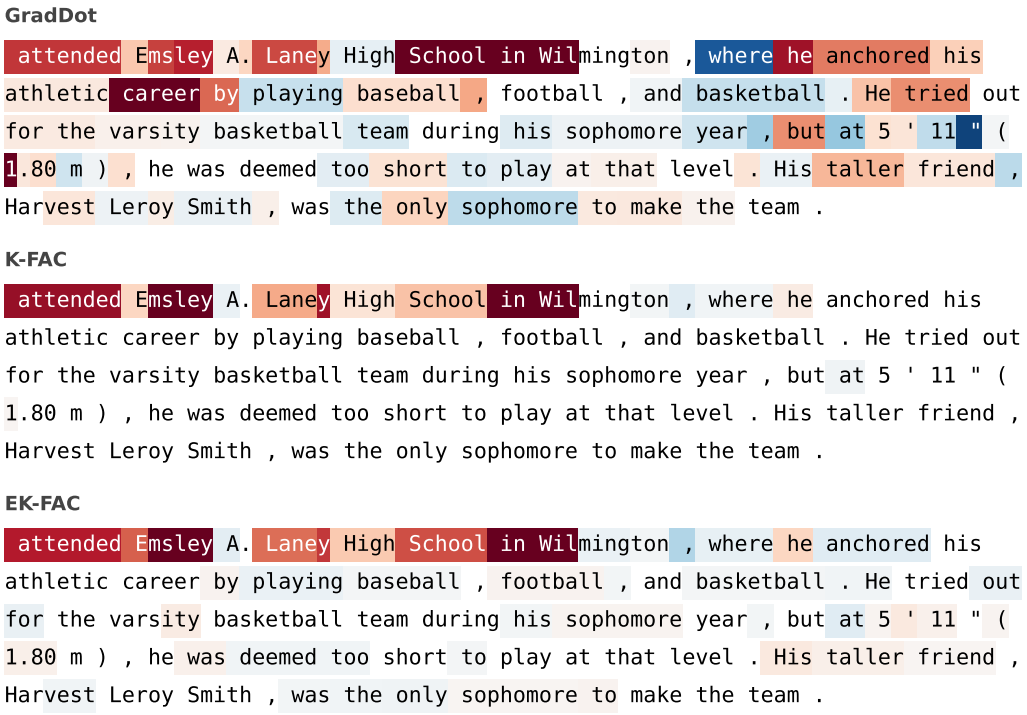

Figure 9: **jordan** (train row 1724). Target: "Michael Jordan attended Emsley A. Laney High School in → **Wilmington**". The paragraph states the fact. GRADDOT ranks `Wil` second, behind the function word *in*, and shades the whole paragraph; K-FAC and EK-FAC put `Wil` first, followed by *in* and the remaining subwords of the city name, and leave the text after the school name essentially blank.

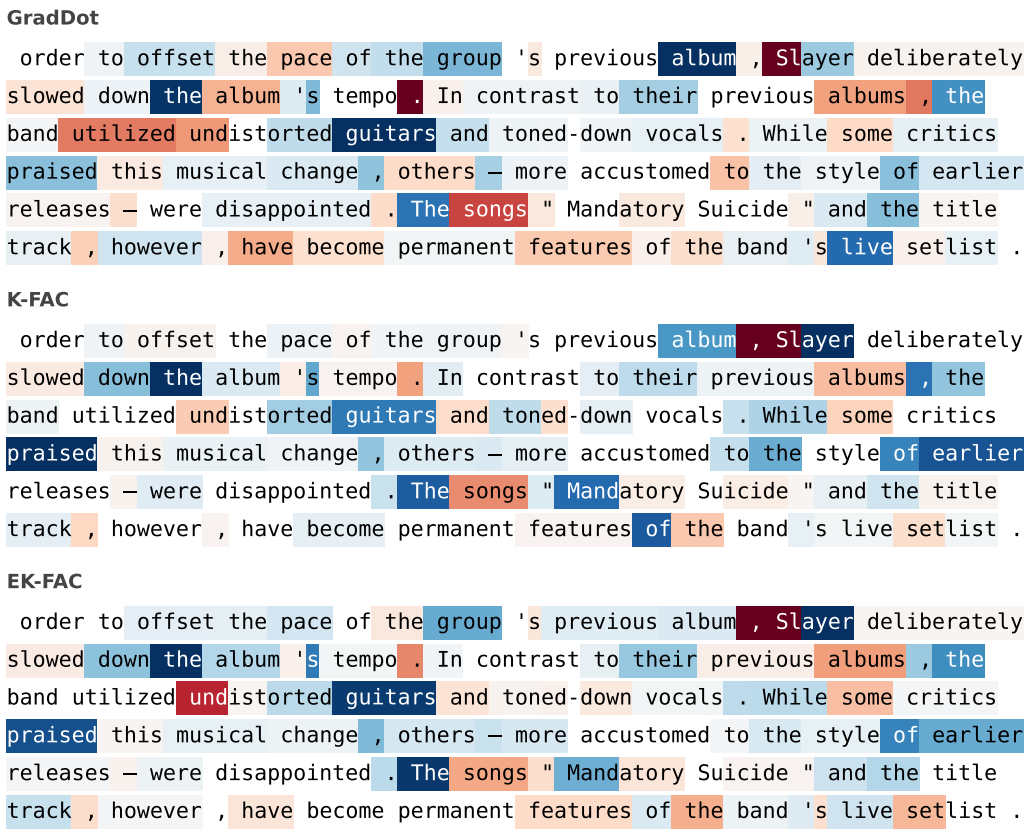

Figure 10: **slayer** (train row 752). Target: "South of Heaven is a studio album by the American thrash metal band → **Slayer**" (tokenized `Slayer`). The answer's first subword ranks first under all three attributors, but its lead over the runner-up grows from 3.6× (GradDot) to 7.3× (EK-FAC) and 35.7× (K-FAC). The paragraph discusses the album's tempo rather than restating the band's name, and the sequence-level GradDot score is in fact negative (−82), whereas both preconditioned scores are positive: preconditioning concentrates positive attribution on the answer token, while the unpreconditioned sequence-level score is negative.

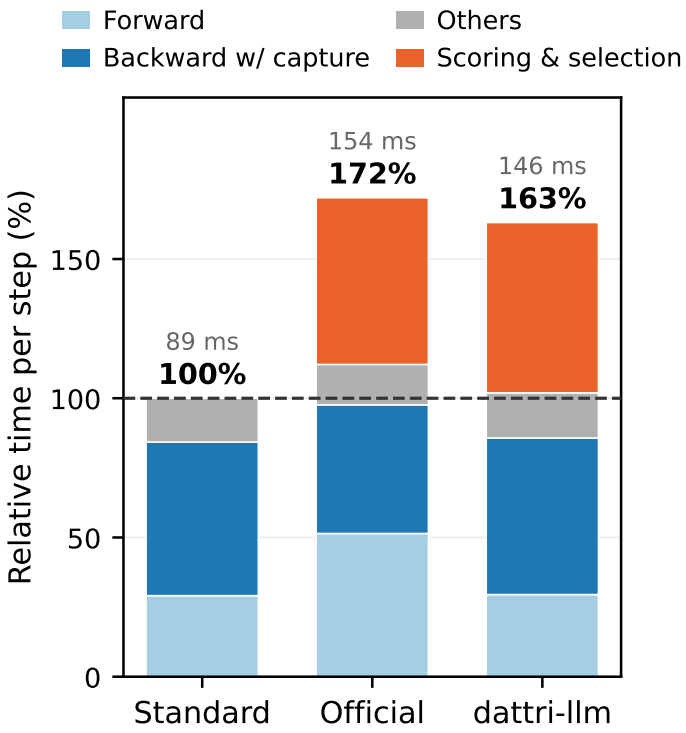

Figure 11: Per-step runtime of online data selection relative to standard training, under full-parameter fine-tuning.