Module-Aware Optimization for Graph Neural Networks

Guy Hadad
Ben-Gurion University of the Negev
Beer-Sheva, Israel
guyhada@post.bgu.ac.il

Haggai Roitman
Ben-Gurion University of the Negev
Beer-Sheva, Israel
haggair@gmail.com

Moshe Eliasof
Ben-Gurion University of the Negev
Beer-Sheva, Israel
eliasof@bgu.ac.il

Bracha Shapira
Ben-Gurion University of the Negev
Beer-Sheva, Israel
bshapira@bgu.ac.il

Abstract

Different optimizers impose different geometries on parameter updates. Adaptive methods such as Adam and AdamW use coordinate-wise updates, while matrix-aware optimizers such as Muon utilize weight-matrix structure through orthogonalized updates. For Graph Neural Networks (GNNs), however, the effective geometry of its weight matrices is shaped not only by its learned channel-mixing weights, but also by the graph-propagated representations it acts on. Thus, even if an update is well normalized as a matrix, its effect on the layer output can be amplified when applied to graph-propagated features. To address this gap, we introduce GraphMuon, a graph-aware optimizer that adapts Muon's matrix updates to the covariance structure of graph-propagated layer inputs. GraphMuon performs orthogonalization in a metric defined by these propagated representations and normalizes step sizes according to the update's induced effect on the layer representation. We show that theoretically, vanilla Muon can suffer from graph-induced update amplification, while our GraphMuon can mitigate this effect. As such, our GraphMuon yields a graph-metric analogue of Muon's polar update step. Empirically, GraphMuon achieves $1.32\times$–$2.42\times$ validation-loss time-to-target speedups over AdamW on three large-scale OGB benchmarks. Across 15 dataset–architecture configurations spanning node classification, inductive graph classification, and graph regression, it achieves lower time-to-target in 13 cases, while maintaining competitive or superior predictive performance. Code is available at https://github.com/guy-hadad/GraphMuon.

1 Introduction

Optimizers define the geometry in which neural-network updates are measured. Classical SGD uses a global Euclidean scale [26], adaptive methods such as Adam and AdamW normalize updates coordinate-wise [17, 20], and preconditioned methods such as Shampoo exploit matrix statistics to approximate richer curvature information [10]. More recently, matrix-aware optimizers such as Muon treat weight matrices as structured operators and replace raw momentum updates with approximately orthogonalized matrix directions [16]. Muon has also been shown to improve compute efficiency in large language-model training [19, 29, 30]. These developments suggest a broader principle: the geometry used by an optimizer should depend not only on the shape of a parameter tensor, but also on the module in which that parameter appears [1].

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

Graph neural networks provide a setting where this module dependence is especially important. In a standard feed-forward layer, a weight matrix acts on representations produced by previous neural layers. In a GNN layer, however, the trainable matrix acts on graph-propagated representations. For a propagation operator S and node representation H, the effective input to the weight matrix is

$$X_g = SH. \tag{1}$$

Therefore, the effect of an update ΔW is governed by the induced representation-level perturbation $X_g \Delta W$, rather than by the parameter-space norm of ΔW alone. Since X_g depends on the graph topology, neighborhood aggregation, feature smoothing, and the current training state, two updates with the same Euclidean matrix scale can induce very different changes in the layer output.

This observation exposes a gap in existing optimizers for GNNs. AdamW adapts updates through coordinate-wise gradient moments, but does not explicitly distinguish directions that are amplified by graph propagation from directions that are attenuated in the propagated feature distribution. Muon improves over coordinate-wise updates by using the matrix structure of W, but its orthogonalization is still input-agnostic: the same Euclidean matrix geometry is used whether the matrix acts on dense features, tokens, or graph-propagated node representations. For GNNs, the relevant object is not only the standalone matrix W, but the graph-conditioned map

$$W \mapsto X_g W. \tag{2}$$

The geometry of this map is determined by the covariance of the propagated inputs,

$$C_g = \frac{1}{n} X_g^\top X_g. \tag{3}$$

Directions aligned with high-energy eigenspaces of C_g produce large representation-level perturbations, while directions in low-energy eigenspaces have much smaller immediate effect [12]. Thus, an optimizer for GNN weight matrices should be both matrix-aware and graph-aware: it should exploit the matrix structure of W, while calibrating update direction and scale to the graph-induced geometry of the representations on which W actually acts.

Inspired by this observation, we introduce **GraphMuon**, a graph-aware extension of Muon for GNN optimization. GraphMuon adapts Muon's matrix-orthogonalized updates to the layer-wise geometry induced by graph-propagated inputs. It has two complementary components. First, *graph-aware orthogonalization* applies the polar-style Muon update in a metric defined by the covariance of X_g. Second, *graph-aware step normalization* scales each update by its induced representation-level perturbation, rather than by its parameter-space magnitude alone. Together, these mechanisms align the optimizer geometry with the actual forward map of a GNN layer.

Our analysis in Section 4.3 shows that this modification is principled: a vanilla Muon update, although normalized in Euclidean matrix space, can still be amplified in representation space by the dominant eigenspaces of C_g. In contrast, our GraphMuon directly controls the quantity $\|X_g \Delta W\|_F / \sqrt{n}$, which is the perturbation induced in the pre-activation representation of the GNN layer. We further show that the graph-aware orthogonalization step is the graph-metric analogue of Muon's polar update, rather than an ad hoc rescaling.

Figure 1 illustrates the resulting mismatch and our correction. A Muon update can be well normalized as a matrix while still causing a large graph-propagated perturbation. GraphMuon addresses this by using C_g to adapt both the update geometry and its scale.

We evaluate GraphMuon across graph-learning benchmarks spanning node classification, inductive graph classification, and graph regression, using GCN, GAT, GINE, and GraphGPS architectures. Under a fixed validation-loss time-to-target protocol, GraphMuon achieves $1.32\times$–$2.42\times$ wall-clock speedups over AdamW on the three large-scale OGB benchmarks. Across a broader evaluation of 15 dataset–architecture configurations, GraphMuon improves epoch-level time-to-target in all cases and achieves faster wall-clock convergence in 13 of 15, while maintaining competitive or superior predictive performance.

Main Contributions. Our contributions are as follows:

1. We identify a *graph-induced optimization mismatch*: standard optimizers normalize updates in parameter space, whereas the effect of a GNN weight update is determined by its perturbation of graph-propagated representations.

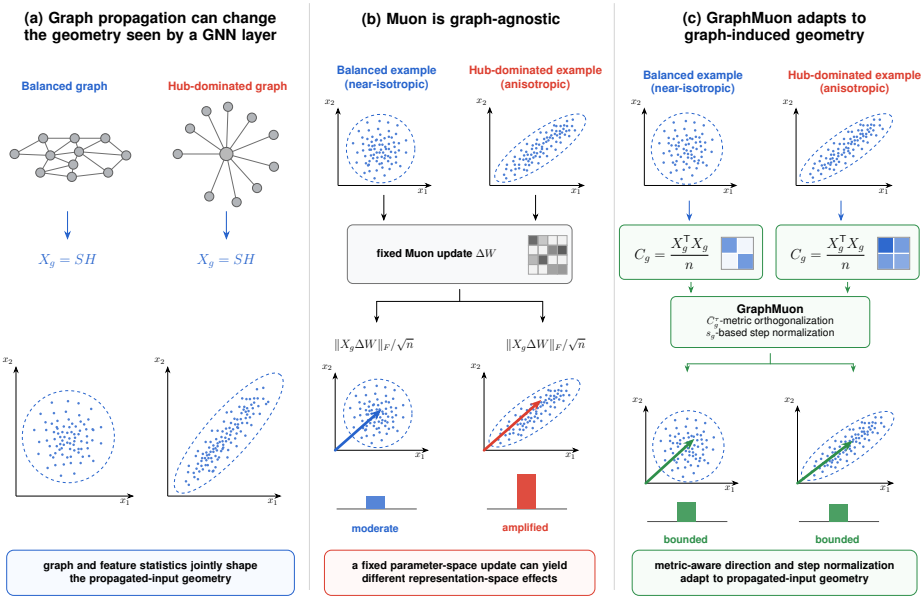


Figure 1: Motivation for graph-aware optimization. A GNN weight matrix acts on the graph-propagated input $X_g = SH$, whose geometry depends on graph structure and propagated feature statistics. As a result, a Muon update that is well normalized in Euclidean matrix space can still produce an amplified representation-level perturbation $\|X_g \Delta W\|_F$. GraphMuon addresses this mismatch by adapting the update geometry to the graph-induced covariance $C_g = X_g^\top X_g/n$ and scaling each step by its induced effect on graph-propagated representations.

2. We introduce GraphMuon, a graph-aware extension of Muon that adapts matrix-update geometry to the layer-wise covariance of graph-propagated representations and normalizes updates by their representation-level effect.
3. We prove that GraphMuon controls graph-induced representation perturbations and yields a graph-metric analogue of Muon's polar update.
4. We demonstrate faster validation-loss convergence across diverse graph-learning regimes, including inductive graph-level tasks and multiple GNN architectures, with wall-clock improvements in 13 of 15 evaluated configurations and competitive or superior predictive performance.

2 Related Work

Graph neural networks. GNNs extend deep learning to graph-structured data by interleaving learnable transformations with neighborhood aggregation [9, 27]. Scalable spatial methods, including GCN [18], GraphSAGE [11], GAT [32], and GIN [33], have become the dominant paradigm, with subsequent work extending these ideas to heterogeneous graphs [28] and heterophilous graphs where connected nodes tend to differ in class [36]. Across these architectures, weight matrices act not on raw node features but on representations smoothed and mixed by graph propagation. This distinguishes GNN training from standard deep-network optimization: the geometry seen by each weight matrix is shaped not only by network depth, but also by graph topology and its induced propagation operator, which is precisely the mismatch GraphMuon is designed to address.

Matrix-aware optimization. Classical SGD and momentum update parameters in Euclidean space, while AdaGrad, Adam, and AdamW normalize updates coordinate-wise using accumulated gradient statistics [4, 17, 20]. Structured optimizers exploit richer geometry: K-FAC uses Kronecker-factored curvature [22], Shampoo uses matrix preconditioners [10], and Muon applies momentum followed by approximate matrix orthogonalization [16]. Muon is the closest predecessor, but remains module-agnostic: it uses the same Euclidean matrix geometry for dense features, tokens, and graph-propagated representations. GraphMuon instead tests whether update geometry should match the module-induced

operator, which for GNNs is the graph-conditioned map $W \mapsto X_g W$, rather than the standalone matrix W.

Graph-aware modules and optimization. A central theme in GNN research is adapting neural-network components to graph structure: GCNs [18], GAT [32], and GIN [33] do so architecturally, as do graph-specific normalization and modular designs [3, 25]. Izadi et al. [15] show that optimization itself can be graph-aware by deriving natural-gradient updates for graph-based semi-supervised learning. GraphMuon follows this optimizer-level view but targets a different setting: rather than a natural-gradient method, it adapts Muon's matrix-aware geometry to the layer-wise covariance of graph-propagated representations, the optimizer-level analogue of graph-adaptive GNN design. Additional related work is discussed in Appendix B.

3 Background and Preliminaries

Notations. Let $G = (V, E)$ be a graph with adjacency matrix $A \in \mathbb{R}^{n \times n}$ and node representations $H^{(\ell)} \in \mathbb{R}^{n \times d_\ell}$. Adding self-loops and symmetrically normalizing gives:

$$S = \widetilde{D}^{-1/2} \widetilde{A} \, \widetilde{D}^{-1/2}, \qquad \widetilde{A} = A + I,$$

where $\widetilde{D}$ is the degree matrix of $\widetilde{A}$. Each GNN weight matrix $W^{(\ell)}$ acts on the *graph-propagated* representation $X_g^{(\ell)} = SH^{(\ell)}$, so the geometry seen by $W^{(\ell)}$ is shaped by both the node features and the graph structure in S.

Muon. Muon [16] is an optimizer for matrix-valued parameters. Given gradient $G_t = \nabla_W \mathcal{L}(W_t)$, it maintains momentum $B_t = \mu B_{t-1} + G_t$ and forms $M_t = G_t + \mu B_t$ (Nesterov) or $M_t = B_t$ otherwise. It replaces M_t with an approximately orthogonalized direction via K Newton–Schulz (NS) iterations,

$$D_t = \mathrm{NS}_K(M_t), \tag{4}$$

and updates $W_{t+1} = W_t - \eta D_t$ (optionally with decoupled weight decay; see Appendix A.3). The key property is that D_t has more balanced singular values than M_t. However, vanilla Muon treats W as a generic matrix and does not account for the graph-induced geometry of $X_g^{(\ell)}$. For additional background, see Appendix A.

4 GraphMuon

In this section we formalize GraphMuon - a Muon-based optimizer specialized for GNNs. Because $W^{(\ell)}$ acts on graph-propagated features, i.e., $X_g^{(\ell)} = SH^{(\ell)}$, the impact of an update depends on the graph-induced feature geometry. We adapt Muon in two ways: *graph-aware orthogonalization* (Section 4.1) adjusts the update direction using this geometry, and *graph-aware step normalization* (Section 4.2) controls the resulting representation-level perturbation. We conclude this section with a discussion on GraphMuon's theoretical properties (Section 4.3).

4.1 Graph-aware Orthogonalization

For layer ℓ, define the graph-induced covariance $C_g^{(\ell)} = \frac{1}{n} X_g^{(\ell)\top} X_g^{(\ell)}$, where $1/n$ removes dependence on graph size. Since $C_g^{(\ell)}$ may be singular or ill-conditioned, we use the trace-normalized and regularized matrix:

$$\bar{C}_g^{(\ell)} = \frac{C_g^{(\ell)}}{\mathrm{tr}(C_g^{(\ell)})/d_\ell} + \epsilon_{\mathrm{cov}} I, \tag{5}$$

where d_ℓ is the input dimension of $W^{(\ell)}$ and $\epsilon_{\mathrm{cov}} > 0$ is a small regularizer. We compute $(\bar{C}_g^{(\ell)})^{\tau/2}$ via eigendecomposition $\bar{C}_g^{(\ell)} = V \Lambda V^\top$, caching eigenbases and recomputing only when $\|\Delta \bar{C}_g^{(\ell)}\|_F > \delta$. For $d_\ell > d_{\max}$, we substitute the diagonal approximation $\mathrm{diag}(\bar{c}_i^{\tau/2})$, reducing cost from $O(d_\ell^3)$ to $O(d_\ell)$.

Graph-aware orthogonalization maps the momentum into the graph metric, orthogonalizes, then maps back:

$$D_{g,\tau,t}^{(\ell)} = \left(\bar{C}_g^{(\ell)}\right)^{-\tau/2} \mathrm{NS}_K\left(\left(\bar{C}_g^{(\ell)}\right)^{\tau/2} M_t^{(\ell)}\right). \tag{6}$$

The hyperparameter $\tau \in [0, 1]$ controls graph-awareness strength. In particular, $\tau = 0$ implements standard Muon, while larger τ increasingly respects the graph-propagated feature geometry. The sensitivity of GraphMuon to this hyperparameter is studied in Appendix F.1.

4.2 Graph-aware Step Normalization

Because $W^{(\ell)}$ acts on $X_g^{(\ell)}$, update size should be measured by the induced representation perturbation $\|X_g^{(\ell)}\Delta W^{(\ell)}\|_F$, not by $\|\Delta W^{(\ell)}\|_F$ alone. We use the *raw* (not trace-normalized) covariance $C_g^{(\ell)}$ to preserve the true layer-wise feature scale (see Appendix D). Given update direction $D^{(\ell)}$, GraphMuon computes the graph-aware scale:

$$s_g^{(\ell)} = \sqrt{\frac{\mathrm{tr}\left(D^{(\ell)\top} C_g^{(\ell)} D^{(\ell)}\right)}{\|D^{(\ell)}\|_F^2} + \epsilon_{\mathrm{scale}}}, \tag{7}$$

where $\epsilon_{\mathrm{scale}} > 0$ is a small constant. Since $\mathrm{tr}(D^{(\ell)\top} C_g^{(\ell)} D^{(\ell)}) = \frac{1}{n}\|X_g^{(\ell)} D^{(\ell)}\|_F^2$, the scale $s_g^{(\ell)}$ can be computed directly from graph-propagated features without forming $C_g^{(\ell)}$.

Following the original Muon implementation, we apply the shape-dependent correction $r = \max(1,\, d_\ell/d_{\ell+1})^{1/2}$, which compensates for the scale change introduced by a semi-orthogonal update on rectangular matrices. The GraphMuon update is given by:

$$W_{t+1}^{(\ell)} = \big(1 - \eta\, r\lambda\big) W_t^{(\ell)} - \eta\, r\, \frac{D^{(\ell)}}{s_g^{(\ell)}}, \tag{8}$$

where η is the learning rate and λ is the decoupled weight-decay coefficient.

4.3 Theoretical Properties

We summarize the main theoretical properties of GraphMuon: it controls the perturbation induced in the representation space actually used by each GNN layer, and its graph-aware orthogonalization is the natural graph-metric analogue of Muon's polar update. All proofs are in Appendix E.

For the realized GNN module $W \mapsto SHW$, the covariance $C_g = H^\top S^\top SH/n$ is the exact pullback of the normalized output-space geometry, whereas $C_H = H^\top H/n$ describes the pre-propagation map $W \mapsto HW$. Thus, graph propagation changes the induced geometry through $S^\top S$, while the underlying module-aware principle may also extend beyond GNNs; further theoretical motivation is provided in Appendix E.

Representation-level perturbation. The key motivation is that, in GNNs, update scale should be measured by the perturbation induced after graph propagation, not by parameter-space magnitude alone. Let D denote the update direction and $\alpha = \eta\, r(d_\ell, d_{\ell+1})$ the scalar multiplying D in Eq. (8). The quantity governing how an update perturbs a layer's pre-activation is $\frac{1}{\sqrt{n}}\|X_g \Delta W\|_F$, not the parameter-space norm $\|\Delta W\|_F$. For a vanilla Muon step $\Delta W_{\mathrm{Muon}} = -\alpha D$,

$$\frac{1}{\sqrt{n}}\,\|X_g \Delta W_{\mathrm{Muon}}\|_F \;\le\; \alpha\sqrt{\lambda_{\max}(C_g)}\,\|D\|_F. \tag{9}$$

Thus, even a matrix-normalized Muon direction can be amplified by the dominant eigenvalue of the graph-induced covariance.

Graph-aware step control. The normalization in Eq. (7) removes this amplification.

Theorem 1 (Graph-representation perturbation control)**.** *For any nonzero direction D, the GraphMuon step $\Delta W_{\mathrm{Graph}} = -\alpha D/s_g$ satisfies:*

$$\frac{1}{\sqrt{n}}\,\|X_g \Delta W_{\mathrm{Graph}}\|_F \;\le\; \alpha\|D\|_F. \tag{10}$$

If $\epsilon_{\mathrm{scale}} = 0$ and $X_g D \neq 0$, equality holds.

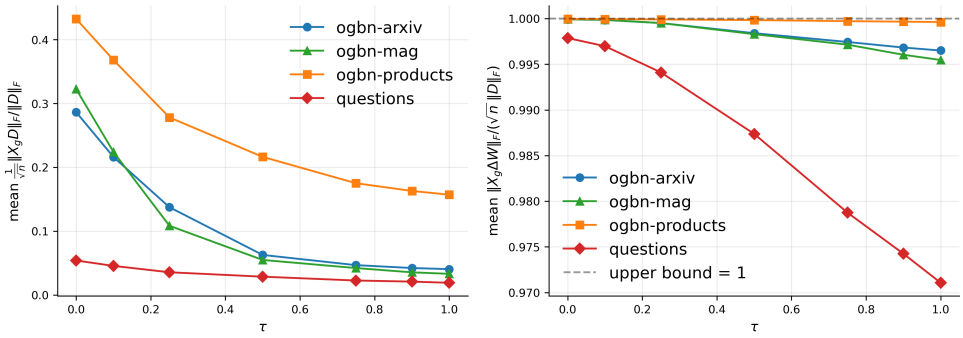


Figure 2: Empirical validation of GraphMuon's representation-level perturbation control. **Left**: before graph-aware normalization, Euclidean-orthogonalized directions induce graph-dependent perturbations whose magnitude varies across datasets and values of τ. **Right**: after scaling by s_g, the normalized perturbation is tightly controlled near the theoretical bound from Theorem 1, confirming that GraphMuon calibrates updates by their effect on X_g rather than by Euclidean parameter-space scale alone.

Unlike Muon, which normalizes in Euclidean matrix space, GraphMuon rescales each direction by its induced effect on the graph-propagated representations, so two directions with equal Frobenius norm produce explicitly calibrated layer-level perturbations, as empirically validated in Figure 2.

Sharpness of Muon's graph amplification. The $\sqrt{\lambda_{\max}(C_g)}$ factor in Eq. (9) is not a loose artifact of the analysis: there exist valid polar directions for which it is attained up to dimension-dependent constants.

Proposition 1 (Sharpness of Muon graph amplification)**.** *Assume $d_\ell \geq d_{\ell+1}$, and let $\lambda_1 \geq \cdots \geq \lambda_{d_\ell}$ be the eigenvalues of C_g. There exists a matrix $D^\star$ with orthonormal columns (a feasible polar-factor direction) such that:*

$$\frac{1}{\sqrt{n}} \|X_g D^\star\|_F = \sqrt{\sum_{i=1}^{d_{\ell+1}} \lambda_i} \;\geq\; \sqrt{\lambda_{\max}(C_g)}. \tag{11}$$

When $d_{\ell+1} = 1$, the $\sqrt{\lambda_{\max}(C_g)}$ amplification is attained exactly.

Proposition 1 shows that Muon's Euclidean polar constraint alone does not prevent graph-amplified representation updates. Such directions are feasible under Muon's geometry and are not explicitly controlled, motivating the graph-aware normalization in GraphMuon.

Figure 3 therefore supports the central motivation for GraphMuon: controlling the Euclidean geometry of a matrix update does not necessarily control its induced perturbation after graph propagation.

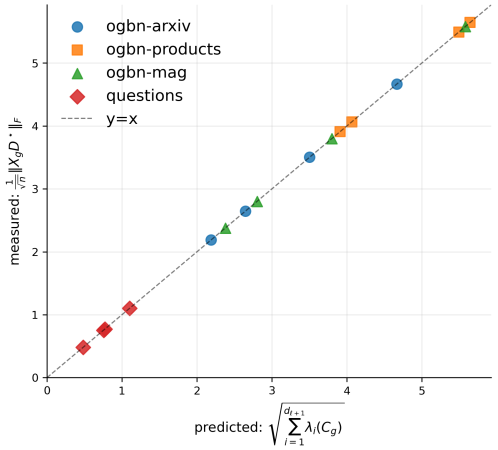


Figure 3: Empirical sharpness of Muon's graph-induced amplification. For each dataset and layer, we construct the polar direction $D^\star$ from the top eigenspaces of the raw graph-induced covariance $C_g = X_g^\top X_g/n$. The measured amplification, $\|X_g D^\star\|_F/\sqrt{n}$, matches the spectral prediction $\left(\sum_{i=1}^{d_{\ell+1}} \lambda_i(C_g)\right)^{1/2}$. The near-diagonal agreement confirms that the amplification term in Proposition 1 is attainable on real graph-propagated representations, rather than a loose worst-case bound.

Graph-metric orthogonalization. The preceding results focus on update scale. We now justify the graph-aware orthogonalization step in Eq. (6). Vanilla Muon can be interpreted as applying a polar projection in Euclidean matrix geometry. GraphMuon replaces this with the corresponding polar projection under the graph-induced metric. We analyze the ideal exact-polar version of the update, replacing the Newton–Schulz approximation with the exact polar factor $P(\cdot)$.

Theorem 2 (Graph-metric polar optimality)**.** *Assume $d_\ell \geq d_{\ell+1}$, $\bar{C}_g \succ 0$, and $\bar{C}_g^{\tau/2} M$ has full column rank. The exact graph-aware direction $D_{g,\tau} = \bar{C}_g^{-\tau/2} P\left(\bar{C}_g^{\tau/2} M\right)$ satisfies*

$$D_{g,\tau}^\top \bar{C}_g^\tau D_{g,\tau} = I, \tag{12}$$

and solves the graph-metric Procrustes problem

$$D_{g,\tau} = \arg \max_{D^\top \bar{C}_g^\tau D = I} \operatorname{tr}\left(D^\top \bar{C}_g^\tau M\right). \tag{13}$$

Theorem 2 shows that the graph-aware direction is not an ad hoc modification of Muon. It is the polar/Muon update obtained after replacing Euclidean orthogonality by graph-metric orthogonality. τ interpolates between vanilla Muon at $\tau = 0$ and graph-metric orthogonalization as τ increases.

Main Takeaways. Our theoretical results show that GraphMuon modifies Muon in two principled ways. Step normalization controls the representation-level perturbation $X_g \Delta W$, while graph-aware orthogonalization computes a polar direction under the graph-induced metric. In contrast, vanilla Muon balances updates in Euclidean parameter space and does not explicitly account for the graph-propagated geometry seen by each GNN layer. The complete layer-wise update is provided in Appendix C, with additional empirical analyses of the theoretical properties in Appendix F.

5 Experiments

In this section, we conduct an extensive set of experiments to demonstrate the effectiveness of GRAPHMUON as an optimizer that adapts GNN updates to graph-propagated representations. Concretely, our experiments address the following questions:

(Q1) Does graph-conditioned optimization improve over parameter-space optimization? We compare GRAPHMUON with AdamW, Muon, Shampoo, and graph-aware natural-gradient baselines to test whether adapting update direction and scale to the graph-propagated covariance improves the main optimization outcome: quicker optimization that maintains competitive or superior test accuracy. This question captures the central distinction of GRAPHMUON: it does not merely change the optimizer family, but changes the geometry in which GNN weight updates are measured.
(Q2) Which component of GRAPHMUON is responsible for the gains? We ablate graph-aware orthogonalization and graph-aware step normalization to determine if improvements come from changing the update direction, controlling the representation-level step size, or their combination.
(Q3) Do empirical diagnostics support the theoretical motivation? We measure representation-level perturbations and graph-induced spectral quantities to test whether Euclidean-normalized matrix updates can indeed be amplified by graph-propagated covariance, and if GRAPHMUON controls it.

5.1 Experimental Setup

Datasets. We evaluate GraphMuon across nine graph-learning benchmarks, covering transductive node classification, heterophilous node classification, inductive graph classification, and graph regression. The node-classification datasets include OGBN-ARXIV, OGBN-PRODUCTS, and OGBN-MAG [14], together with QUESTIONS [24], Cora, and Citeseer. To assess generalization beyond transductive node prediction, we additionally evaluate on the molecular graph-classification benchmarks OGBG-MOLBACE and OGBG-MOLTOX21, and the graph-regression benchmark ZINC. Additional dataset information is provided in Appendix G.1.

Backbones. We evaluate multiple GNN architectures, including GCN, GAT [31], GINE, and GraphGPS [25]. GCN provides the primary comparison on the large-scale OGB node-classification benchmarks. GAT and GraphGPS extend the evaluation to learned attention-based propagation, while GINE is used for molecular graph-level tasks. Additional experiments examine GraphGCN and deeper GNNs. Within each dataset–architecture configuration, all optimizers use the same backbone, training budget, and evaluation protocol, isolating the optimizer as the varying component. Architectural details are provided in Appendix G.2.

Baselines. The primary validation-loss time-to-target comparison includes AdamW [20] and Muon [16]. AdamW is a strong adaptive first-order baseline with decoupled weight decay, while

Table 1: Validation-loss time-to-target across graph-learning tasks. The target is the minimum mean validation loss achieved by AdamW for each dataset–architecture configuration. Entries report epochs / cumulative wall-clock seconds to reach the target. Speedup is relative to AdamW; higher is better. Bold indicates the fastest optimizer. "–" denotes failure to reach the target within the training budget.

Dataset	Backbone	Target loss	AdamW (ep / s)	Muon (ep / s)	GraphMuon (ep / s)	Speedup
Large-scale node classification						
OGBN-ARXIV	GCN	0.8753	433 / 30.3	188 / 16.6	**164 / 13.1**	2.32×
OGBN-MAG	GCN	2.6930	100 / 58.3	65 / 38.1	**41 / 24.1**	2.42×
OGBN-PRODUCTS	GCN	0.3182	300 / 1663.5	–	**226 / 1257.3**	1.32×
Heterophilous node classification						
QUESTIONS	GAT	0.1217	481 / 15.2	320 / 10.2	**167 / 7.1**	2.14×
Inductive graph classification						
OGBG-MOLBACE	GINE	0.6212	65 / 59.1	–	**22 / 29.8**	1.98×
OGBG-MOLTOX21	GINE	0.2946	9 / 22.3	6 / 22.6	**4 / 18.6**	1.20×
Graph regression						
ZINC	GINE	0.2502	293 / 946.6	242 / 1038.4	**106 / 679.0**	1.39×

Table 2: Generalization across GNN architectures and tasks. Results are test performance (mean ± standard deviation over five seeds). Higher is better for accuracy and ROC-AUC; lower is better for MAE. Bold indicates the best result.

Dataset / Backbone	Metric	AdamW	Muon	GraphMuon
ogbn-arxiv / GAT	Acc. ↑	$\mathbf{.6928 \pm .0027}$	$.6911 \pm .0021$	$.6926 \pm .0027$
ogbn-mag / GraphGPS	Acc. ↑	$.2939 \pm .0018$	$.3250 \pm .0032$	$\mathbf{.3296 \pm .0050}$
Tox21 / GINE	ROC-AUC ↑	$.6633 \pm .0223$	$.6553 \pm .0074$	$\mathbf{.6644 \pm .0175}$
BACE / GINE	ROC-AUC ↑	$.7127 \pm .0570$	$.7810 \pm .0194$	$\mathbf{.7927 \pm .0429}$
ZINC / GINE	MAE ↓	$.2303 \pm .01742$	$.1929 \pm .00213$	$\mathbf{.1897 \pm .00112}$

Muon is the closest matrix-aware baseline, applying approximate orthogonalization to momentum updates. We also consider structured preconditioning with Shampoo [10] and graph-aware natural-gradient optimization [15], with implementation details provided in Appendix G.3.

Evaluation. We follow the corresponding benchmark evaluation protocols and report predictive performance using test accuracy for node classification, ROC-AUC for molecular classification, and MAE for graph regression. For optimization efficiency, we adopt a fixed validation-loss time-to-target protocol. For each dataset–architecture configuration, the target is the minimum mean validation loss attained by AdamW across seeds. We measure the first epoch and corresponding cumulative wall-clock training time at which each optimizer's mean validation-loss trajectory reaches this target. Wall-clock speedup is defined as the ratio of AdamW's time-to-target to that of the compared optimizer. Configurations that do not reach the target within the training budget are marked as "–". Additional evaluation results and per-configuration timings are provided in Appendix G.

5.2 Main Results

For the primary comparison, we report validation-loss time-to-target across the large-scale OGB benchmarks, heterophilous node classification, and inductive graph-level tasks. Table 1 shows that GraphMuon reaches AdamW's validation-loss target faster across all seven reported configurations.

On the three large-scale OGB benchmarks, GraphMuon achieves wall-clock speedups of $2.32\times$ on OGBN-ARXIV, $2.42\times$ on OGBN-MAG, and $1.32\times$ on OGBN-PRODUCTS. The corresponding 95% bootstrap confidence intervals are $[2.01, 2.69]$, $[2.29, 2.53]$, and $[1.19, 1.37]$, respectively. These results demonstrate faster convergence to a fixed validation-loss target rather than to a potentially variable accuracy threshold.

The improvements also extend beyond transductive GCNs. On QUESTIONS with GAT, GraphMuon achieves a $2.14\times$ wall-clock speedup. On the inductive molecular graph-classification benchmarks OGBG-MOLBACE and OGBG-MOLTOX21, it achieves $1.98\times$ and $1.20\times$ speedups, respectively. On ZINC with GINE, GraphMuon reaches the validation-loss target $1.39\times$ faster than AdamW, demonstrating applicability to graph regression.

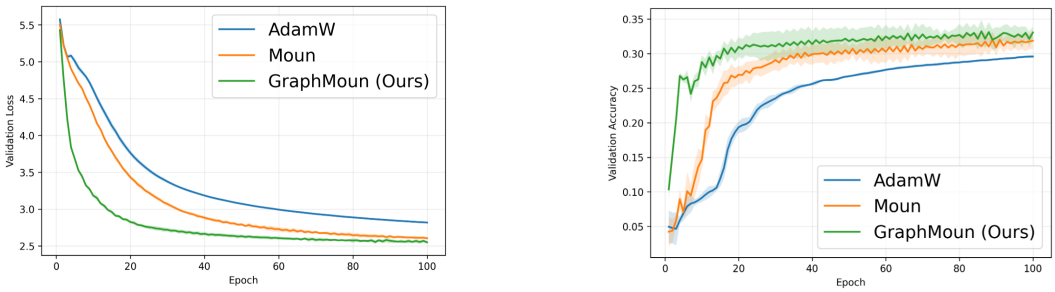


Figure 4: Training dynamics on OGBN-MAG. The left panel shows validation loss and the right panel shows validation accuracy over training. GRAPHMUON makes faster progress in both metrics: it reduces validation loss earlier and reaches strong validation accuracy in fewer epochs than AdamW and Muon, suggesting that graph-induced update geometry can accelerate GNN learning.

Table 3: Ablation of GraphMuon components on OGB benchmarks. Speedup is measured relative to vanilla Muon under a fixed rounded-down Muon validation target; higher is better.

Dataset↓ / Method →	Muon	+ Graph-aware Orthog.	+ Graph-aware Step Norm.	GraphMuon (Ours)
ogbn-arxiv	$1.00\times$	$5.39 \pm 0.31\times$	$8.34 \pm 0.85\times$	$\mathbf{9.02 \pm 0.80}\times$
ogbn-mag	$1.00\times$	$1.07 \pm 0.04\times$	$2.28 \pm 0.22\times$	$\mathbf{3.60 \pm 0.12}\times$
ogbn-products	$1.00\times$	$1.03 \pm 0.02\times$	$1.51 \pm 0.05\times$	$\mathbf{1.90 \pm 0.06}\times$

Additional experiments with GAT, GINE, GraphGPS, and GraphGCN are reported in Appendix G.8. Across all 15 dataset–architecture configurations, GraphMuon reaches the target in fewer epochs than AdamW in every case and achieves the lowest wall-clock time-to-target in 13 of 15 cases, with an arithmetic-mean wall-clock speedup of $4.62\times$. These findings indicate that adapting the optimizer to graph-propagated representation geometry improves optimization efficiency across different architectures and graph-learning tasks.

Overall, GraphMuon validates our central understanding: normalizing updates in the graph-propagated layer geometry improves GNN time-to-target without sacrificing competitive final accuracy. Figure 4 further shows that these gains come from faster optimization dynamics.

Generalization across architectures and tasks. To evaluate generalization beyond GCN-based node classification, we extend the evaluation to GAT, GINE, and GraphGPS, covering node classification, inductive graph classification, and graph regression. As shown in Table 2, GraphMuon achieves the best numerical performance in four of five configurations and remains competitive with AdamW on GAT. The improvements on GINE and GraphGPS demonstrate that the method also applies to graph-level prediction and learned attention-based architectures. Additional validation-loss time-to-target results across 15 configurations are provided in Appendix G.8.

5.3 Ablation Study

We ablate the two main components of GraphMuon: graph-aware orthogonalization and graph-aware step normalization. To focus on optimization speed, we use a relaxed time-to-target metric. For each dataset, the target is the best validation loss reached by vanilla Muon, rounded down to two decimal places (e.g., $0.723 \mapsto 0.72$). This threshold is close to the converged Muon performance while avoiding sensitivity to small validation fluctuations. We then measure the wall-clock time required to first reach this threshold and report speedup relative to vanilla Muon.
Table 3 reports the main ablation on the OGB benchmarks, where the relaxed target is not saturated and therefore distinguishes the contribution of each component. The QUESTIONS ablation is omitted from the main table because all graph-aware variants reach the rounded Muon target after a single optimization step, making the time-to-target metric uninformative for component comparison.

6 Conclusions

We motivate our GraphMuon by a representation-space view of optimization in GNNs. For a layer with graph-propagated input $X_g = SH$, the effective update size is not $\|\Delta W\|_F$, but the induced perturbation $\|X_g \Delta W\|_F/\sqrt{n}$. Our analysis shows that a Euclidean-normalized Muon update can still be amplified by the graph-induced covariance $C_g = X_g^\top X_g/n$, through its dominant eigenspaces. GraphMuon addresses this by normalizing the update according to its induced representation-level effect and by replacing Euclidean matrix geometry with a graph-metric analogue.

This theory also explains when GraphMuon may provide less benefit. The method is most useful when C_g is stable, anisotropic, and informative. On very small graphs, however, C_g may be low-rank and noisy, especially when the number of nodes is small relative to the hidden dimension. In this regime, the graph-induced metric can be poorly estimated, and controlling $\|X_g \Delta W\|_F$ does not necessarily imply faster convergence or better generalization. AdamW can therefore remain competitive, since its coordinate-wise moment estimates do not rely on estimating a graph-level covariance.

This limitation is particularly relevant for small inductive graph-level tasks, where each graph contains few node representations. In such settings, the main optimization challenge may be dominated by sparse local features, atom or bond types, and mini-batch variability rather than by graph-induced update amplification. A natural direction for future work is to adapt the strength of graph awareness to the reliability of C_g, for example, using effective rank, spectral anisotropy, or covariance stability to interpolate between vanilla Muon and fully graph-aware updates.

Acknowledgments and Disclosures

ME acknowledges support from the the Israel Science Foundation through a personal grant (ISF 3062/26) and an equipment grant (ISF 3066/26), as well as support from the Israeli Ministry of Innovation, Science & Technology. The authors declare no competing interests.

References

[1] Jeremy Bernstein. Modular manifolds. *Thinking Machines Lab: Connectionism*, 2025. doi: 10.64434/tml.20250926. https://thinkingmachines.ai/blog/modular-manifolds/.

[2] Rajendra Bhatia. *Matrix Analysis*, volume 169 of *Graduate Texts in Mathematics*. Springer, New York, 1997.

[3] Tianle Cai, Shengjie Luo, Keyulu Xu, Di He, Tie-yan Liu, and Liwei Wang. Graphnorm: A principled approach to accelerating graph neural network training. In *International Conference on Machine Learning*, pages 1204–1215. PMLR, 2021.

[4] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[5] Moshe Eliasof, Beatrice Bevilacqua, Carola-Bibiane Schönlieb, and Haggai Maron. Granola: Adaptive normalization for graph neural networks. *Advances in Neural Information Processing Systems*, 37:90514–90551, 2024.

[6] Moshe Eliasof, Alessio Gravina, Andrea Ceni, Claudio Gallicchio, Davide Bacciu, and Carola-Bibiane Schönlieb. Graph adaptive autoregressive moving average models. In *Forty-second International Conference on Machine Learning*, 2025.

[7] Ky Fan, Alan J Hoffman, et al. Some metric inequalities in the space of matrices. In *Proc. Amer. Math. Soc*, volume 6, pages 111–116. World Scientific, 1955.

[8] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[9] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 2017.

[10] Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization. In *International Conference on Machine Learning*, pages 1842–1850. PMLR, 2018.

[11] William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, volume 30, pages 1024–1034, 2017.

[12] Roger A. Horn and Charles R. Johnson. *Matrix Analysis*. Cambridge University Press, 2 edition, 2012.

[13] W Hu, B Liu, J Gomes, M Zitnik, P Liang, V Pande, and J Leskovec. Strategies for pre-training graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2020.

[14] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

[15] Mohammad Rasool Izadi, Yihao Fang, Robert Stevenson, and Lizhen Lin. Optimization of graph neural networks with natural gradient descent. In *2020 IEEE international conference on big data (big data)*, pages 171–179. IEEE, 2020.

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

[17] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[18] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[19] Jingyuan Liu, Jianlin Su, Xingcheng Yao, Zhejun Jiang, Guokun Lai, Yulun Du, Yidao Qin, Weixin Xu, Enzhe Lu, Junjie Yan, et al. Muon is scalable for llm training. *arXiv preprint arXiv:2502.16982*, 2025.

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

[21] Krishna S Mantri, Xinzhi Wang, Carola-Bibiane Schönlieb, Bruno Ribeiro, Beatrice Bevilacqua, and Moshe Eliasof. Digraf: Diffeomorphic graph-adaptive activation function. *Advances in Neural Information Processing Systems*, 37:3649–3681, 2024.

[22] James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature. In *International conference on machine learning*, pages 2408–2417. PMLR, 2015.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[24] Oleg Platonov, Denis Kuznedelev, Michael Diskin, Artem Babenko, and Liudmila Prokhorenkova. A critical look at the evaluation of gnns under heterophily: Are we really making progress? *arXiv preprint arXiv:2302.11640*, 2023.

[25] Ladislav Rampášek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. *Advances in Neural Information Processing Systems*, 35:14501–14515, 2022.

[26] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

[27] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[28] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *The Semantic Web:15th International Conference, ESWC 2018*, pages 593–607. Springer, 2018.

[29] Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[30] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

[31] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[32] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[33] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[34] Keyulu Xu, Mozhi Zhang, Stefanie Jegelka, and Kenji Kawaguchi. Optimization of graph neural networks: Implicit acceleration by skip connections and more depth. In *International conference on machine learning*, pages 11592–11602. PMLR, 2021.

[35] Chenxiao Yang, Qitian Wu, David Wipf, Ruoyu Sun, and Junchi Yan. How graph neural networks learn: Lessons from training dynamics. In *International Conference on Machine Learning*, pages 56594–56623. PMLR, 2024.

[36] Jiong Zhu, Yujun Yan, Lingxiao Zhao, Mark Heimann, Leman Akoglu, and Danai Koutra. Beyond homophily in graph neural networks: Current limitations and effective designs. In *Advances in Neural Information Processing Systems*, volume 33, pages 7793–7804, 2020.

A Background

Here we provide complete definitions and implementation details for the methods referenced in the main paper. We give the standard GCN layer formulation, the Adam and AdamW update rules, and the full Muon algorithm including the Newton–Schulz iteration and its practical implementation details.

A.1 GCN Layer Definition

Given the symmetrically normalized adjacency S (defined in Section 3), a standard GCN layer computes

$$H^{(\ell+1)} = \sigma\Big(SH^{(\ell)}W^{(\ell)}\Big), \tag{14}$$

where $W^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell+1}}$ is the trainable weight matrix and $\sigma(\cdot)$ is a nonlinear activation. The degree matrix entries are

$$\widetilde{D}_{ii} = \sum_{j=1}^{n} \widetilde{A}_{ij} = 1 + \sum_{j=1}^{n} A_{ij}.$$

A.2 Adam and AdamW

Let θ_t be the parameters and $g_t = \nabla_\theta \mathcal{L}(\theta_t)$ the stochastic gradient. Adam maintains first and second moment estimates:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \tag{15}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)(g_t \odot g_t). \tag{16}$$

With bias correction, the update is

$$\theta_{t+1} = \theta_t - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}. \tag{17}$$

AdamW decouples weight decay:

$$\theta_{t+1} = (1-\eta\lambda)\theta_t - \eta \frac{\widehat{m}_t}{\sqrt{\widehat{v}_t} + \epsilon}. \tag{18}$$

Adam-based optimizers operate coordinate-wise, normalizing each parameter independently, and do not exploit matrix structure or graph-induced geometry.

A.3 Muon: Full Details

The polar factor of a matrix M is

$$\mathcal{P}(M) = M(M^\top M)^{-1/2}, \tag{19}$$

with the analogous expression $(MM^\top)^{-1/2}M$ when $d_{\text{in}} < d_{\text{out}}$. Muon approximates this via Newton–Schulz iterations. Starting from

$$X_0 = \frac{M_t}{\|M_t\|_F + \epsilon}, \tag{20}$$

it applies the quintic iteration for $k = 0, \ldots, K-1$:

$$A_k = X_k X_k^\top, \tag{21}$$

$$X_{k+1} = aX_k + bA_kX_k + cA_k^2X_k, \tag{22}$$

with coefficients

$$a = 3.4445, \qquad b = -4.7750, \qquad c = 2.0315, \tag{23}$$

selected via an ad-hoc gradient-based search in the original implementation [16] to rapidly approximate the polar factor. When $d_{\text{in}} > d_{\text{out}}$, M_t is transposed before the iteration (so the squared matrix is formed over the smaller dimension) and transposed back afterward, reducing the dominant matrix-multiplication cost. After $K = 5$ iterations the direction is $D_t = \mathrm{NS}_K(M_t)$. The vanilla Muon update with decoupled weight decay is

$$W_{t+1} = (1-\eta\lambda)W_t - \eta D_t. \tag{24}$$

Algorithm 1 Full GRAPHMUON update for GNN weight matrices

Require: Learning rate η, momentum μ, weight decay λ, Newton–Schulz steps K, graph-awareness strength τ, regularization constants ϵ_{cov} and $\epsilon_{\mathrm{scale}}$; normalized adjacency S; layer representations $\{H^{(\ell)}\}_\ell$; parameters $\{W_t^{(\ell)}\}_\ell$; momentum buffers $\{B_{t-1}^{(\ell)}\}_\ell$

Ensure: Updated parameters $\{W_{t+1}^{(\ell)}\}_\ell$

for each matrix-valued GNN layer ℓ **do**

$G_t^{(\ell)} \leftarrow \nabla_{W^{(\ell)}} \mathcal{L}_t$ {gradient}

$B_t^{(\ell)} \leftarrow \mu B_{t-1}^{(\ell)} + G_t^{(\ell)}$ {momentum buffer}

$M_t^{(\ell)} \leftarrow G_t^{(\ell)} + \mu B_t^{(\ell)}$ {Nesterov momentum matrix}

$X_g^{(\ell)} \leftarrow S H^{(\ell)}$ {graph-propagated layer input}

$C_g^{(\ell)} \leftarrow \frac{1}{n} X_g^{(\ell)\top} X_g^{(\ell)}$ {graph-induced covariance}

$\bar{C}_g^{(\ell)} \leftarrow C_g^{(\ell)} / \left(\mathrm{tr}(C_g^{(\ell)})/d_\ell\right) + \epsilon_{\mathrm{cov}} I$ {trace-normalize and regularize}

$A_\pm^{(\ell)} \leftarrow \mathrm{MatPow}\left(\bar{C}_g^{(\ell)}, \tau/2\right)$ {covariance half-powers}

$D^{(\ell)} \leftarrow A_-^{(\ell)} \,\mathrm{NS}_K\left(A_+^{(\ell)} M_t^{(\ell)}\right)$ {graph-aware Muon direction}

$s_g^{(\ell)} \leftarrow \sqrt{\mathrm{tr}\left(D^{(\ell)\top} C_g^{(\ell)} D^{(\ell)}\right) / \|D^{(\ell)}\|_F^2 + \epsilon_{\mathrm{scale}}}$ {graph-aware scale}

$\alpha_\ell \leftarrow \eta\, r(d_\ell, d_{\ell+1})$ {shape-corrected layerwise step size}

$W_{t+1}^{(\ell)} \leftarrow (1 - \alpha_\ell \lambda)\, W_t^{(\ell)} - \alpha_\ell D^{(\ell)} / s_g^{(\ell)}$ {decoupled weight decay and update}

end for

return $\{W_{t+1}^{(\ell)}\}_\ell$

B Additional Related Work

Graph-adaptive neural components. A broad line of work adapts standard neural-network components to graph-structured data. GCNs use normalized adjacency propagation [18], GAT adapts attention to graph neighborhoods [32], and GIN studies aggregation from the perspective of expressive power [33]. Graph-specific design has also been extended to normalization and modular architectures. GraphNorm introduces graph-specific normalization [3], GRANOLA uses neighborhood-structure-based normalization [5], GraphGPS combines local message passing with global attention and positional encodings [25], GRAMA [6] learns a graph-adaptive autoregressive moving average filtering technique, and DiGRAF learns graph-adaptive activation functions [21]. These works support the broader principle that generic deep-learning components often benefit from graph-specific adaptation. GraphMuon applies this principle to optimization rather than architecture.

GNN training dynamics. Several works study why GNN optimization differs from optimization in standard Euclidean architectures. Natural-gradient methods exploit information-geometric structure in graph-based semi-supervised learning [15]; skip connections, depth, and label distributions have been shown to affect GNN convergence [34]; and recent training-dynamics analyses show that gradient descent can implicitly exploit graph structure through kernel–graph alignment [35]. These results motivate our representation-space view of optimization: the effect of a weight update in a GNN is determined not only by the update matrix itself, but by the graph-propagated representations on which it acts.

C GRAPHMUON Full Algorithm

Algorithm 1 gives the complete GRAPHMUON update for matrix-valued GNN weight matrices. The algorithm follows the update defined in Section 4: for each layer, it computes the graph-propagated input, forms the corresponding graph-induced covariance, applies graph-aware orthogonalization to the Muon momentum direction, and rescales the resulting update by its induced representation-level perturbation. Non-matrix parameters, such as biases or normalization parameters, can be updated with the accompanying base optimizer.

D Step Normalization: Use of Raw vs. Normalized Covariance

The trace-normalized covariance $\bar{C}_g^{(\ell)}$ used in Eq. (6) removes the layer-wise average feature energy so that orthogonalization depends only on the *anisotropy* of the graph-propagated features. Step normalization has the opposite goal: it should measure the actual representation-level perturbation induced by the update. Since

$$\operatorname{tr}\Big(D^{(\ell)\top} C_g^{(\ell)} D^{(\ell)}\Big) = \frac{1}{n}\left\| X_g^{(\ell)} D^{(\ell)} \right\|_F^2,$$

using the raw $C_g^{(\ell)}$ makes $s_g^{(\ell)}$ equal (up to ϵ_{scale}) to the RMS pre-activation perturbation per column of $X_g^{(\ell)} D^{(\ell)}$. Substituting $\bar{C}_g^{(\ell)}$ would divide out the average feature energy and remove precisely the layer-wise scale that the normalization is intended to control.

E Proofs of Theoretical Results

Throughout this appendix we drop the layer superscript (ℓ) and write $\alpha = \eta\, r(d_\ell, d_{\ell+1})$ for the effective per-layer step size. We denote by $d_{\text{in}} = d_\ell$ and $d_{\text{out}} = d_{\ell+1}$ the input and output dimensions of the weight matrix $W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$.

E.1 Shared Setup

Graph-induced covariance. Given graph-propagated features $X_g = SH \in \mathbb{R}^{n \times d_{\text{in}}}$, define the graph-induced covariance

$$C_g \;=\; \frac{1}{n} X_g^\top X_g \;\in\; \mathbb{R}^{d_{\text{in}} \times d_{\text{in}}}. \tag{25}$$

C_g is positive semidefinite. Let $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_{d_{\text{in}}} \geq 0$ denote its eigenvalues and $V = [v_1, \ldots, v_{d_{\text{in}}}]$ the corresponding orthonormal eigenvectors, so that $C_g = V \Lambda V^\top$ with $\Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_{d_{\text{in}}})$.

Key identity. For any matrix $\Delta W \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$,

$$\frac{1}{n}\left\| X_g \Delta W \right\|_F^2 = \frac{1}{n} \operatorname{tr}\big(\Delta W^\top X_g^\top X_g\, \Delta W\big) = \operatorname{tr}\big(\Delta W^\top C_g\, \Delta W\big). \tag{26}$$

Spectral bound. For any positive semidefinite C and any matrix V with columns v_j,

$$\operatorname{tr}(V^\top C V) = \sum_j v_j^\top C v_j \;\leq\; \lambda_{\max}(C) \sum_j \|v_j\|_2^2 = \lambda_{\max}(C)\, \|V\|_F^2. \tag{27}$$

Module-induced geometry. Consider the realized GNN module $F_g(W) = SHW$, with S and H fixed at the current step. Its Jacobian maps a parameter perturbation ΔW to $SH\Delta W$. Consequently, the normalized Frobenius geometry of the module output induces the parameter-space inner product

$$\frac{1}{n}\langle SH\Delta W_1, SH\Delta W_2\rangle_F = \operatorname{tr}(\Delta W_1^\top C_g \Delta W_2), \qquad C_g = \frac{1}{n} H^\top S^\top S H. \tag{28}$$

In contrast, the pre-propagation map $W \mapsto HW$ induces $C_H = H^\top H/n$, with

$$C_g - C_H = \frac{1}{n} H^\top (S^\top S - I) H. \tag{29}$$

The two metrics coincide when propagation is isometric on the feature subspace, including $S = I$. Otherwise, C_H does not capture the amplification or attenuation introduced by graph propagation.

Connection to the optimization objective. Let $Z = SHW$, and assume the loss $\mathcal{L}$ is locally β-smooth in Z under the normalized output norm $\|U\|_{\text{out}} = \|U\|_F/\sqrt{n}$. Then

$$\begin{aligned} \mathcal{L}(Z + SH\Delta W) \leq \mathcal{L}(Z) &+ \langle \nabla_Z \mathcal{L}, SH\Delta W\rangle_F \\ &+ \frac{\beta}{2} \operatorname{tr}(\Delta W^\top C_g \Delta W). \end{aligned} \tag{30}$$

Thus, $\|SH\Delta W\|_F/\sqrt{n}$ is precisely the perturbation magnitude governing this local smoothness bound. A parameter-space norm alone cannot distinguish directions with different effects on the realized module: for any unit eigenvector v_i of C_g with eigenvalue λ_i and any unit vector q, the perturbation $\Delta W_i = v_i q^\top$ satisfies $\|\Delta W_i\|_2 = 1$, while $\|SH\Delta W_i\|_F^2/n = \lambda_i$. This motivates controlling representation-level perturbations rather than relying exclusively on Euclidean normalization of the parameter update.

Classical orthogonal Procrustes result.

Lemma 1 (Orthogonal Procrustes, [7])**.** *Let $A \in \mathbb{R}^{m\times k}$ with $m \geq k$ and $A = U\Sigma V^\top$ its (thin) SVD. Then*

$$\max_{\substack{Q\in\mathbb{R}^{m\times k}\\ Q^\top Q = I}} \operatorname{tr}(Q^\top A) = \operatorname{tr}(\Sigma), \tag{31}$$

and the unique maximiser (assuming A has full column rank) is the polar factor $P(A) = UV^\top$.

This classical result (see also [2]) is used in the proof of Theorem 2.

E.2 Proof of Theorem 1 (Graph-Representation Perturbation Control)

Proof. The GraphMuon step is $\Delta W_{\text{Graph}} = -\alpha D/s_g$, where

$$s_g = \sqrt{\frac{\operatorname{tr}(D^\top C_g D)}{\|D\|_F^2} + \epsilon_{\text{scale}}}. \tag{32}$$

Step 1: apply the trace identity. By (26),

$$\frac{1}{n}\big\|X_g\,\Delta W_{\text{Graph}}\big\|_F^2 = \frac{\alpha^2}{s_g^2}\,\operatorname{tr}(D^\top C_g D). \tag{33}$$

Step 2: bound $1/s_g^2$. Since $\epsilon_{\text{scale}} \geq 0$, the definition (32) gives

$$s_g^2 = \frac{\operatorname{tr}(D^\top C_g D)}{\|D\|_F^2} + \epsilon_{\text{scale}} \;\geq\; \frac{\operatorname{tr}(D^\top C_g D)}{\|D\|_F^2}, \tag{34}$$

and therefore

$$\frac{1}{s_g^2} \;\leq\; \frac{\|D\|_F^2}{\operatorname{tr}(D^\top C_g D)}. \tag{35}$$

Step 3: combine. Substituting (35) into (33),

$$\frac{1}{n}\big\|X_g\,\Delta W_{\text{Graph}}\big\|_F^2 \;\leq\; \frac{\alpha^2\|D\|_F^2}{\operatorname{tr}(D^\top C_g D)}\cdot \operatorname{tr}(D^\top C_g D) = \alpha^2\,\|D\|_F^2. \tag{36}$$

Taking square roots establishes the inequality $\frac{1}{\sqrt{n}}\|X_g\,\Delta W_{\text{Graph}}\|_F \leq \alpha\|D\|_F$.

Equality case. When $\epsilon_{\text{scale}} = 0$, the bound (35) holds with equality, so every inequality above is an equality provided the common factor $\operatorname{tr}(D^\top C_g D)$ is nonzero. We have $\operatorname{tr}(D^\top C_g D) = \frac{1}{n}\|X_g D\|_F^2$, which is zero if and only if $X_g D = 0$. Hence, when $\epsilon_{\text{scale}} = 0$ and $X_g D \neq 0$, the inequality holds with equality. □

E.3 Proof of Proposition 1 (Sharpness of Muon Graph Amplification)

Proof. Recall the eigendecomposition $C_g = V\Lambda V^\top$ with eigenvalues $\lambda_1 \geq \cdots \geq \lambda_{d_{\text{in}}} \geq 0$ and orthonormal eigenvectors $v_1, \ldots, v_{d_{\text{in}}}$.

Construction of $D^\star$. Define

$$D^\star = \big[v_1 \mid v_2 \mid \cdots \mid v_{d_{\text{out}}}\big] \;\in\; \mathbb{R}^{d_{\text{in}}\times d_{\text{out}}}. \tag{37}$$

Since the eigenvectors are orthonormal, $D^{\star\top} D^\star = I_{d_{\text{out}}}$, so $D^\star$ has orthonormal columns. A matrix with orthonormal columns is the polar factor of itself, making $D^\star$ a feasible polar-factor direction under Muon's geometry.

Representation energy of $D^\star$. Applying the trace identity (26) and the eigendecomposition,

$$\begin{aligned}\frac{1}{n}\big\|X_g D^\star\big\|_F^2 &= \mathrm{tr}(D^{\star\top} C_g\, D^\star) \\ &= \mathrm{tr}\Big([v_1 \cdots v_{d_{\mathrm{out}}}]^\top V \Lambda V^\top [v_1 \cdots v_{d_{\mathrm{out}}}]\Big) \\ &= \sum_{i=1}^{d_{\mathrm{out}}} v_i^\top V \Lambda V^\top v_i = \sum_{i=1}^{d_{\mathrm{out}}} \lambda_i. \end{aligned} \tag{38}$$

Taking square roots,

$$\frac{1}{\sqrt{n}}\big\|X_g D^\star\big\|_F = \sqrt{\sum_{i=1}^{d_{\mathrm{out}}} \lambda_i}. \tag{39}$$

Comparison with $\sqrt{\lambda_{\max}(C_g)}$. Because $\lambda_1 = \lambda_{\max}(C_g)$ and all $\lambda_i \geq 0$,

$$\sum_{i=1}^{d_{\mathrm{out}}} \lambda_i \;\geq\; \lambda_1 = \lambda_{\max}(C_g), \tag{40}$$

so $\frac{1}{\sqrt{n}}\|X_g D^\star\|_F \geq \sqrt{\lambda_{\max}(C_g)}$.

Case $d_{\mathrm{out}} = 1$. When $d_{\mathrm{out}} = 1$, we have $D^\star = v_1$ and $\frac{1}{\sqrt{n}}\|X_g D^\star\|_F = \sqrt{\lambda_1} = \sqrt{\lambda_{\max}(C_g)}$, so the amplification is attained exactly.

Implication for Muon. The vanilla Muon bound in (9) reads $\frac{1}{\sqrt{n}}\|X_g \Delta W_{\mathrm{Muon}}\|_F \leq \alpha\sqrt{\lambda_{\max}(C_g)}\|D\|_F$. The construction above shows that $D^\star$, which is feasible under Muon's Euclidean polar constraint ($D^{\star\top} D^\star = I$), achieves $\frac{1}{\sqrt{n}}\|X_g D^\star\|_F = \sqrt{\sum_{i=1}^{d_{\mathrm{out}}} \lambda_i} \geq \sqrt{\lambda_{\max}(C_g)}$. Hence, the dependence on $\lambda_{\max}(C_g)$ cannot be removed by Muon's matrix-normalization alone, confirming that the bound is not a loose artifact. □

Moreover, for any positive diagonal matrix Σ, the momentum matrix $M = D^\star\Sigma$ has polar factor $P(M) = D^\star$, confirming that the construction corresponds to a feasible exact Muon update.

E.4 Proof of Theorem 2 (Graph-Metric Polar Optimality)

Proof. Throughout, let $\tilde{M} = \bar{C}_g^{\tau/2} M$ and $Q = P(\tilde{M})$ be the polar factor of $\tilde{M}$, so that $D_{g,\tau} = \bar{C}_g^{-\tau/2} Q$ by definition. Since $\bar{C}_g \succ 0$ and $\tilde{M}$ has full column rank (by assumption), Q is unique and satisfies $Q^\top Q = I_{d_{\mathrm{out}}}$.

Part 1: $D_{g,\tau}^\top \bar{C}_g^\tau D_{g,\tau} = I$.

Direct computation:

$$\begin{aligned} D_{g,\tau}^\top\, \bar{C}_g^\tau\, D_{g,\tau} &= \big(\bar{C}_g^{-\tau/2} Q\big)^\top \bar{C}_g^\tau \big(\bar{C}_g^{-\tau/2} Q\big) \\ &= Q^\top \bar{C}_g^{-\tau/2}\, \bar{C}_g^\tau\, \bar{C}_g^{-\tau/2}\, Q \\ &= Q^\top Q \\ &= I. \end{aligned} \tag{41}$$

Here we used that $\bar{C}_g \succ 0$ so its matrix powers commute and $\bar{C}_g^{-\tau/2}\bar{C}_g^\tau\bar{C}_g^{-\tau/2} = I$.

Part 2: $D_{g,\tau}$ solves the graph-metric Procrustes problem.

We wish to show

$$D_{g,\tau} = \operatorname*{arg\,max}_{\substack{D \in \mathbb{R}^{d_{\mathrm{in}} \times d_{\mathrm{out}}} \\ D^\top \bar{C}_g^\tau D = I}} \mathrm{tr}\big(D^\top \bar{C}_g^\tau M\big). \tag{42}$$

Change of variables. Let $\tilde{D} = \bar{C}_g^{\tau/2} D$. Since $\bar{C}_g \succ 0$, the map $D \mapsto \tilde{D}$ is a bijection on $\mathbb{R}^{d_{\mathrm{in}} \times d_{\mathrm{out}}}$.

The constraint $D^\top \bar{C}_g^\tau D = I$ transforms as

$$\tilde{D}^\top \tilde{D} = D^\top \bar{C}_g^{\tau/2} \bar{C}_g^{\tau/2} D \cdot (\text{by } \bar{C}_g \succ 0) \; = D^\top \bar{C}_g^\tau D = I. \tag{43}$$

Hence the constraint becomes $\tilde{D}^\top \tilde{D} = I$, i.e., $\tilde{D}$ has orthonormal columns.

The objective transforms as

$$\operatorname{tr}(D^\top \bar{C}_g^\tau M) = \operatorname{tr}(D^\top \bar{C}_g^{\tau/2} \bar{C}_g^{\tau/2} M) = \operatorname{tr}(\tilde{D}^\top \tilde{M}), \tag{44}$$

where $\tilde{M} = \bar{C}_g^{\tau/2} M$.

Apply Lemma 1. The reformulated problem is

$$\max_{\tilde{D}^\top \tilde{D} = I} \operatorname{tr}(\tilde{D}^\top \tilde{M}), \tag{45}$$

which is exactly the orthogonal Procrustes problem. By Lemma 1, the unique maximiser is $\tilde{D}^\star = \mathcal{P}(\tilde{M}) = Q$.

Map back. Returning to the original variable, $D^\star = \bar{C}_g^{-\tau/2} \tilde{D}^\star = \bar{C}_g^{-\tau/2} Q = D_{g,\tau}$, which establishes (42).

Recovery of vanilla Muon at $\tau = 0$. When $\tau = 0$, the covariance factors become $\bar{C}_g^0 = I$, so $\tilde{M} = M$, and $D_{g,0} = P(M)$, which is the exact polar factor used by vanilla Muon. The constraint $D^\top D = I$ is then the standard Euclidean orthonormality constraint, confirming that Theorem 2 generalizes vanilla Muon's polar update. □

Output-space interpretation. The graph-metric Procrustes solution also minimizes the distance to the momentum matrix in the same metric:

$$D_{g,\tau} = \arg \min_{D^\top \bar{C}_g^\tau D = I} \left\| \bar{C}_g^{\tau/2} (D - M) \right\|_F^2. \tag{46}$$

At $\tau = 1$, let $c = \operatorname{tr}(C_g)/d_{\text{in}}$. Since $\bar{C}_g = C_g/c + \epsilon_{\text{cov}} I$, the graph-metric orthogonality constraint implies

$$\frac{1}{n} (X_g D_{g,1})^\top (X_g D_{g,1}) = c \left(I - \epsilon_{\text{cov}} D_{g,1}^\top D_{g,1} \right). \tag{47}$$

In the unregularized full-rank case, the realized update therefore has orthogonal columns of equal squared RMS magnitude. Graph-metric orthogonalization thus transfers Muon's singular-value balancing from parameter space to the module-output space. While this component balances the realized update directions, graph-aware step normalization separately controls their overall representation-level magnitude.

F Additional GRAPHMUON Diagnostics

This appendix provides additional diagnostics for the graph-induced covariance used by GraphMuon. These results are not required for the main correctness claims, but they help interpret when graph-aware optimization is expected to matter and how the graph-awareness parameter τ affects the update geometry.

F.1 Effect of Graph-Awareness Strength

GraphMuon uses the interpolation parameter τ to control the strength of graph-aware orthogonalization. When $\tau = 0$, the covariance factors vanish and the update reduces to vanilla Muon. As τ increases, the update direction increasingly respects the geometry induced by the graph-propagated covariance.

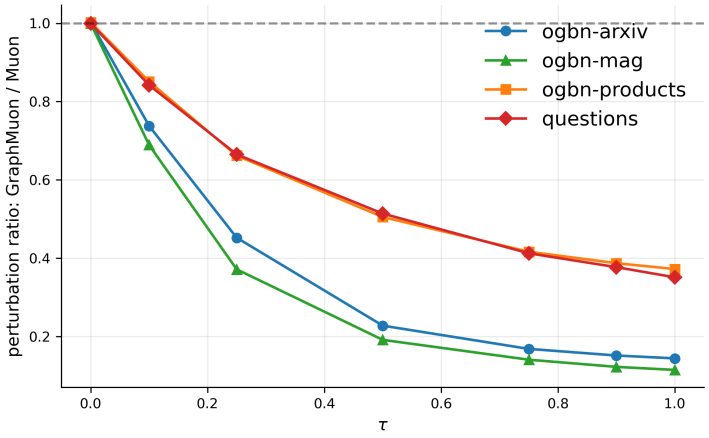
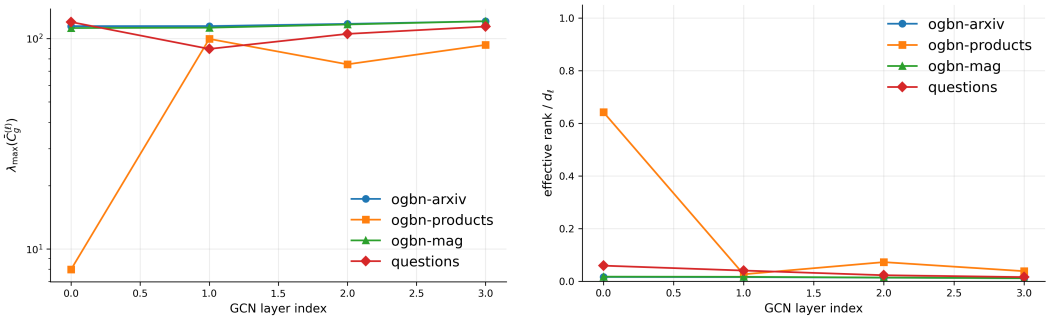


Figure 5: Effect of the graph-awareness parameter τ on representation-level perturbations. The plot reports the perturbation ratio between GraphMuon and vanilla Muon as τ varies. Values below one indicate that the graph-aware direction induces a smaller representation-level perturbation than the corresponding Euclidean Muon direction. Across datasets, increasing τ generally moves the update away from the Euclidean Muon geometry and toward the graph-induced metric, supporting the interpretation of τ as a continuous control over graph awareness.



(a) Dominant eigenvalue.

(b) Normalized effective rank.

Figure 6: Spectral diagnostics of the graph-induced covariance across datasets and layers. Left: the dominant eigenvalue of $\bar{C}_g$ identifies high-energy graph-propagated directions along which Euclidean-normalized updates can induce large representation-level perturbations. Right: the normalized effective rank measures how broadly the energy of $\bar{C}_g$ is distributed across directions. Lower effective rank indicates that the graph-propagated representation is concentrated in fewer dominant modes, making Euclidean matrix geometry less aligned with the actual representation geometry. Together, these diagnostics support the motivation for GraphMuon: the covariance seen by each GNN layer is often anisotropic and layer-dependent, so update normalization should account for the graph-propagated input geometry.

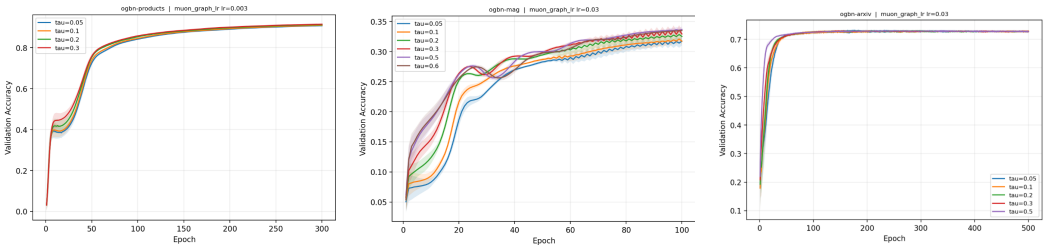
F.2 Spectral Diagnostics of Graph-Induced Covariance

The behavior of GraphMuon depends on the spectral structure of the graph-induced covariance $C_g = X_g^\top X_g/n$. The dominant eigenvalue measures the largest graph-amplified direction, while the effective rank measures how concentrated or distributed the graph-propagated representation is across spectral directions. These diagnostics help explain both the benefits and the limitations of graph-aware optimization.

Controlled spectral analysis. To isolate the interaction between graph propagation and feature alignment, we additionally evaluate GraphMuon on four-block stochastic block model (SBM) graphs with varying alignment between node features and planted communities. Table 4 shows that as alignment weakens, the community Rayleigh score decreases from 0.708 to 0.174, while the planted

Table 4: Spectral analysis on four-block stochastic block model (SBM) graphs with varying feature-community alignment. We report the Rayleigh score of the planted community signal, its contribution to the graph-induced covariance C_g, and the update energy in dominant C_g directions before and after GraphMuon scaling.

Alignment	Community Rayleigh	Planted share of C_g	Top-C_g energy Muon $\rightarrow$ GraphMuon
Strong	0.708	20.3%	2.18% $\rightarrow$ 0.47%
Medium	0.401	9.7%	2.37% $\rightarrow$ 0.43%
Weak	0.174	2.6%	1.06% $\rightarrow$ 0.18%



(a) `ogbn-products` (b) `ogbn-mag` (c) `ogbn-arxiv`

Figure 7: Sensitivity of GraphMuon to the graph-metric exponent τ. Each panel shows validation accuracy as a function of training epoch for several values of τ, with the learning rate fixed within each dataset. GraphMuon is generally robust to τ, although the most effective value is dataset-dependent. On `ogbn-arxiv`, all values achieve nearly identical final accuracy. On `ogbn-products`, moderate values yield slightly faster optimization and marginally better final performance. On `ogbn-mag`, larger values of τ provide clearer gains, suggesting that stronger graph-aware geometry is more beneficial in this setting.

contribution to C_g decreases from 20.3% to 2.6%. Across all three regimes, GraphMuon substantially reduces update energy in the dominant graph-amplified directions relative to Muon. These findings demonstrate that the graph-induced geometry depends jointly on spectral propagation strength and feature alignment, and provide controlled empirical support for graph-aware update normalization.

F.3 Sensitivity to the Graph-Metric Exponent τ

We study the sensitivity of GraphMuon to the graph-metric exponent τ, which controls the strength of the graph-aware metric transformation. Smaller values of τ keep the update geometry closer to vanilla Muon, while larger values place more emphasis on the graph-induced covariance structure. We evaluate several values of τ on representative OGB node-classification benchmarks and plot the validation accuracy over training.

Figure 7 shows that GraphMuon is reasonably robust to the choice of τ, although the best value is somewhat dataset-dependent. On `ogbn-arxiv`, all tested values produce very similar final validation accuracy, with only minor differences in early training speed. On `ogbn-products`, moderate values ($\tau \in [0.1, 0.3]$) yield slightly stronger early progress and marginally better final validation performance than the smallest tested value. On `ogbn-mag`, the effect of τ is more pronounced: larger values generally improve optimization and lead to better validation accuracy, while very small values converge more slowly and to slightly weaker solutions.

Overall, these results indicate that GraphMuon is not overly brittle to the choice of τ, but that tuning τ can provide additional gains, especially on more challenging or structurally heterogeneous datasets such as `ogbn-mag`. In practice, a small grid over τ is sufficient to identify a strong configuration.

G Additional Experimental Setup

G.1 Dataset Statistics

Table 5 reports the statistics of the node-classification benchmarks used in our experiments. These datasets cover homogeneous citation and co-purchase graphs, a heterogeneous academic graph, and a heterophilous benchmark. For all datasets, we follow the official splits and evaluation protocols from the corresponding benchmark release.

Table 5: Dataset statistics for the node-classification benchmarks. For OGBN-MAG, node and edge counts are reported over all node and relation types, while features and targets correspond to the target paper nodes.

Dataset	Graph regime	#Nodes	#Edges	#Features	#Targets
OGBN-ARXIV	Homogeneous citation	169,343	1,166,243	128	40
OGBN-PRODUCTS	Homogeneous co-purchase	2,449,029	61,859,140	100	47
OGBN-MAG	Heterogeneous academic	1,939,743	21,111,007	128	349
QUESTIONS	Heterophilous benchmark	48,921	153,540	301	2

G.2 Architectures

Unless stated otherwise, all experiments follow the architecture, data splits, evaluation protocol, and training setup specified in the original benchmark papers. This design keeps the benchmark setting fixed and isolates the effect of the optimizer being compared [14, 24]. Table 6 summarizes the backbone used for each dataset.

GCN backbone (transductive node classification). We use an L-layer GCN with symmetric normalized propagation. Given the adjacency matrix A with added self-loops, we form $\hat{A} = \tilde{D}^{-1/2}(A + I)\tilde{D}^{-1/2}$ and compute

$$H^{(l+1)} = \sigma\Big(\hat{A}\, H^{(l)}\, W^{(l)}\Big), \qquad l = 0, \ldots, L-2, \tag{48}$$

with a linear readout at layer L. Weight matrices are stored in $\mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ (i.e. the propagated feature matrix multiplies on the right) and initialized with Xavier uniform initialization. Intermediate layers apply ReLU activation, dropout, and residual connections whenever $d_{\text{in}} = d_{\text{out}}$. Default settings are depth $L = 2$, hidden dimension $d = 64$, and dropout rate $p = 0.5$. Layer normalization is disabled unless noted otherwise. The final-layer logits are passed to a cross-entropy loss over the training-mask nodes.

GINE backbone (QUESTIONS). For the heterophilous QUESTIONS benchmark we follow the architecture of Platonov et al. [24] and use a GINE backbone [13] adapted for transductive node classification. Each layer applies the update

$$h_v^{(l+1)} = \mathrm{MLP}^{(l)}\left((1 + \varepsilon^{(l)})\, h_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \mathrm{ReLU}\Big(h_u^{(l)} + e_{uv}\Big)\right), \tag{49}$$

where e_{uv} are learned edge feature embeddings and $\varepsilon^{(l)}$ is a trainable scalar. Node features are projected to the hidden dimension by a linear input encoder; edge features are embedded by a learned QUESTIONS-specific edge encoder. Unlike the molecular GINE setup, no `AtomEncoder`/`BondEncoder` or graph-level pooling is used: the final-layer node embeddings are passed directly through a linear head to produce per-node logits, and a binary cross-entropy loss is computed over the training-mask nodes following the official QUESTIONS protocol. For graph-aware optimizer variants, each GINE layer's first MLP weight matrix is treated as the graph-conditioned operator, with its graph covariance computed from the aggregated GINE input $(1 + \varepsilon^{(l)})\, h_v^{(l)} + \sum_{u \in \mathcal{N}(v)} \mathrm{ReLU}(h_u^{(l)} + e_{uv})$; subsequent MLP weight matrices are treated as standard dense layers and use the post-activation hidden representation as their input.

Heterogeneous graphs. For datasets with heterogeneous structure (e.g. ogbn-mag) we follow the standard preprocessing and backbone established for the corresponding OGB benchmark, including paper-node subgraph extraction for ogbn-mag.

Table 6: Dataset–architecture configurations in the expanded evaluation. The original GCN/GINE settings are retained, with additional architectures and graph-level tasks introduced in the rebuttal.

Dataset	**Task**	**Backbones**
ogbn-arxiv	Node classification	GCN, GAT, GraphGPS
ogbn-mag	Node classification	GCN, GAT, GraphGPS
ogbn-products	Node classification	GCN
Questions	Node classification	GINE, GAT
Cora, Citeseer	Node classification	GCN
ogbg-molbace	Graph classification	GINE
ogbg-moltox21	Graph classification	GINE
ZINC	Graph regression	GINE, GraphGPS, GraphGCN

G.3 Optimizer Hyperparameters and Implementation Details

Hyperparameter selection protocol. All optimizers were tuned using the same validation protocol and the official validation split for each dataset. For each optimizer and dataset, the base learning rate was selected from

$$\{0.001,\ 0.003,\ 0.01,\ 0.03,\ 0.1\,,\ 0.15\}.$$

The configuration with the best validation accuracy was used for test accuracy reporting. The same selected configuration was then used for time-to-target evaluation, without additional retuning for the target metric.

Best-validation versus fastest-target configurations. For optimizer variants reported without a superscript, hyperparameters are selected by the highest validation accuracy on the official validation split. Thus, GRAPHMUON denotes the configuration selected by best validation performance, using the same model-selection criterion as the other optimizers.

Rows marked with a superscript $*$ report the fastest configuration within the same hyperparameter grid. Specifically, GRAPHMUON* denotes the GraphMuon configuration that reaches the AdamW best-validation target in the shortest wall-clock time. This configuration is used only for time-to-target analysis and is reported separately from the best-validation configuration to distinguish predictive model selection from optimization speed selection.

Training budget. For each dataset, we use the training budget specified by the corresponding benchmark implementation. Time-to-target is computed only within these benchmark budgets; runs that do not reach the target within the corresponding budget are marked as not reaching the target.

Table 7: Dataset-specific training budgets used in our experiments.

Dataset	Benchmark source	Training budget
OGBN-ARXIV	OGB official [14]	500 epochs
OGBN-PRODUCTS	OGB official [14]	300 epochs
OGBN-MAG	OGB official [14]	100 epochs
QUESTIONS	Platonov et al. benchmark [24]	300 epochs

Common settings. Unless otherwise noted, all experiments share the hyperparameters in Table 8.

AdamW. AdamW uses $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$, and decoupled weight decay $\lambda = 5 \times 10^{-4}$.

Muon. Muon uses momentum $\mu = 0.95$, Nesterov momentum, and $K = 5$ Newton–Schulz orthogonalization iterations. Matrix-valued weight parameters are updated with Muon. Bias vectors and all non-matrix parameters are updated with AdamW using the same selected base learning rate and $\lambda_{\text{other}} = 0$.

GraphMuon. GraphMuon uses the same momentum, Nesterov, Newton–Schulz, and weight-decay settings as Muon. The graph-awareness parameter is selected from

$$\tau \in \{0.1,\ 0.2,\ 0.3,\ 0.5\}$$

for each dataset using validation accuracy. In our implementation, this corresponds to the `muon_graph_lr` optimizer variant.

Table 8: Shared training hyperparameters.

Hyperparameter	Value
Depth L	2
Hidden dimension d	64
Dropout p	0.5
Weight decay λ	5×10^{-4}
Residual connections	enabled when dimensions match
Layer normalization	disabled
LR scheduler	none
Early-stopping patience	100
Seeds	10

Let $W^{(\ell)} \in \mathbb{R}^{d_{\text{in}} \times d_{\text{out}}}$ denote the weight matrix of layer ℓ, and let $G_t^{(\ell)} = \nabla_{W^{(\ell)}} \mathcal{L}_t$ be its stochastic gradient at step t. For readability, we omit the layer superscript below.

Nesterov momentum. GraphMuon first forms the Muon momentum matrix

$$B_t = \mu B_{t-1} + G_t, \tag{50}$$

$$M_t = G_t + \mu B_t, \tag{51}$$

with $\mu = 0.95$ and $B_0 = 0$.

Graph covariance. For each GNN layer, let

$$H_{\text{prop}} = X_g = SH$$

denote the graph-propagated input to the weight matrix. The raw graph-induced covariance is

$$C_g = \frac{1}{N} H_{\text{prop}}^\top H_{\text{prop}}, \tag{52}$$

when all N propagated node representations are used. For scalability, when $N > m$, a subset $H' \in \mathbb{R}^{m \times d}$ is drawn uniformly without replacement from all N rows of H_{prop} at every optimizer step, using a fresh sample for each update. In this case, we estimate

$$C_g = \frac{1}{m} {H'}^\top H'. \tag{53}$$

Both C_g and $\bar{C}$ are computed from this single draw; no second sample is taken for the step-normalization term.

For graph-metric orthogonalization, we use the trace-normalized and regularized covariance

$$\bar{C} = \frac{C_g}{\text{tr}(C_g)/d_{\text{in}}} + \varepsilon_{\text{metric}} I. \tag{54}$$

A single regularization constant

$$\varepsilon_{\text{metric}} = 10^{-5}$$

is used throughout: it is added to the diagonal of $\bar{C}$ for covariance regularization, used to clamp eigenvalues away from zero when computing fractional matrix powers, and added inside the graph step norm s_g to prevent division by zero. All uses share the same value and are denoted uniformly by $\varepsilon_{\text{metric}}$.

For $d_{\text{in}} \leq 512$, fractional matrix powers of $\bar{C}$ are computed by eigendecomposition. For larger input dimensions, we use the diagonal approximation described in Section 4.1.

Graph-metric orthogonalization. GraphMuon maps the momentum matrix into the graph metric, applies Newton–Schulz orthogonalization, and maps the result back:

$$O_t = \bar{C}^{-\tau/2} \text{NSK}_K \left(\bar{C}^{\tau/2} M_t \right), \tag{55}$$

where $K = 5$ and NSK_K denotes K Newton–Schulz orthogonalization steps. Setting $\tau = 0$ recovers the vanilla Muon orthogonalization direction.

Graph-aware learning-rate scaling. After orthogonalization, GraphMuon rescales the update according to its induced perturbation under the raw graph covariance:

$$s_g = \sqrt{\frac{\mathrm{tr}(O_t^\top C_g O_t)}{\|O_t\|_F^2} + \varepsilon_{\mathrm{metric}}}. \tag{56}$$

Here C_g is the raw, unnormalized covariance from Eq. (52) or Eq. (53), not the trace-normalized covariance $\bar{C}$. Thus, s_g measures the root-mean-square graph-metric energy of the update direction. Directions aligned with high-variance graph-propagated features receive a smaller effective step, while directions aligned with lower-energy graph directions receive a relatively larger step.

Shape-dependent learning-rate correction. Following Muon, we apply the rectangular-matrix correction

$$r(d_{\mathrm{in}}, d_{\mathrm{out}}) = \sqrt{\max\left(1, \frac{d_{\mathrm{in}}}{d_{\mathrm{out}}}\right)}, \tag{57}$$

which compensates for the RMS scale of semi-orthogonal updates.

Parameter update. Weight decay is applied in decoupled form before the gradient step, using the shape-corrected rate and without graph learning-rate scaling:

$$W_{t+1} = \big(1 - \eta\, r\, \lambda\big) W_t - \eta\, r \cdot \frac{O_t}{s_g}, \tag{58}$$

where $r = r(d_{\mathrm{in}}, d_{\mathrm{out}})$, $\lambda = 5 \times 10^{-4}$ is weight decay, and η is the grid-searched base learning rate. Weight decay is applied only to matrix weights; bias vectors and other non-matrix parameters are updated by AdamW with $\lambda_{\mathrm{other}} = 0$.

Shampoo [10]. Shampoo uses momentum 0.0, $\varepsilon = 10^{-4}$, and preconditioner update frequency of one step.

Natural Gradient [15]. The natural-gradient baseline uses Tikhonov regularization $\varepsilon_{\mathrm{kfac}} = 0.01$, covariance update frequency of 50 steps, no running-average smoothing ($\alpha = 1.0$), and π-correction disabled.

G.4 Efficiency of the Graph-Aware Muon Implementation

We describe the implementation mechanisms used to keep the additional cost of GraphMuon modest relative to vanilla Muon. The key design choice is that the optimizer reuses graph-propagated activations already computed during the ordinary forward pass, and only estimates low-dimensional covariance statistics from them.

Activation reuse. For a GCN layer, the forward pass computes the graph-propagated input

$$H_{\mathrm{prop}} = \hat{A} H$$

before applying the learnable weight matrix. Our implementation stores a detached reference to this tensor as `cached_h_prop`. The optimizer then reads this cached tensor at step time. Thus, GraphMuon does not perform an additional forward pass and does not apply an additional graph propagation inside the optimizer. The extra runtime cost at this stage is only the detach/reference operation, while the extra memory cost is the lifetime of the cached H_{prop} tensor until the optimizer step.

Subsampled covariance estimation. Let N denote the number of available node representations and let $m =$ `metric_sample_size` denote the covariance-estimation budget (default $m = 2048$). When $N > m$, we sample m rows of H_{prop} without replacement and form the covariance estimate

$$C_g = \frac{1}{m} H'^\top H',$$

where $H' \in \mathbb{R}^{m \times d_{\mathrm{in}}}$ contains the sampled rows. This reduces the covariance cost from $O(N d_{\mathrm{in}}^2)$ to $O(m d_{\mathrm{in}}^2)$. When $N \le m$, the full set of nodes is used. The same sampled rows are used both for C_g and for the normalized metric $\bar{C}$, so only one sample is drawn per optimizer step.

Full and diagonal covariance paths. When $d_{\mathrm{in}} \le$ `max_full_dim`, GraphMuon forms the full $d_{\mathrm{in}} \times d_{\mathrm{in}}$ covariance and computes its eigendecomposition. For larger input dimensions, the

Table 9: Efficiency-related GraphMuon hyperparameters.

Hyperparameter	Default	Role
`metric_sample_size`	2 048	Maximum number of nodes used to estimate C_g per step. Setting it to 0 disables subsampling and uses all nodes.
`max_full_dim`	512	Input dimension threshold for forming the full $d_{\text{in}} \times d_{\text{in}}$ covariance. Above this threshold, a diagonal approximation is used.
`eigen_refresh_steps`	10	Maximum number of optimizer steps for which a cached eigendecomposition of $\bar{C}$ may be reused. Setting it to ≤ 0 disables caching.
`eigen_change_tol`	0.05	Relative Frobenius-change threshold used to trigger an early refresh of the eigendecomposition.
`torch_foreach`	`true`	Enables PyTorch foreach kernels for weight decay and parameter updates, reducing Python and kernel-launch overhead.

implementation falls back to a diagonal metric. In this case, writing $c = \operatorname{diag}(H'^{\top} H')/m$, we use the diagonal approximation

$$\bar{C}_{\text{diag}} = \operatorname{diag}\left(\frac{c}{\bar{c} + \varepsilon_{\text{metric}}} + \varepsilon_{\text{metric}} \right),$$

where $\bar{c}$ is the mean diagonal entry. This reduces memory from $O(d_{\text{in}}^2)$ to $O(d_{\text{in}})$ and avoids the $O(d_{\text{in}}^3)$ eigendecomposition. Fractional metric powers and graph-aware step normalization are then applied element-wise. In our standard benchmark configurations, $d_{\text{in}} \leq 128$, so the full covariance path is used.

Eigendecomposition caching. For the full covariance path, computing the eigendecomposition of $\bar{C}$ costs $O(d_{\text{in}}^3)$ per layer. GraphMuon caches the eigenvectors and eigenvalues of $\bar{C}$ in the optimizer state and reuses them when the metric has not changed substantially. Specifically, if the cached decomposition is less than `eigen_refresh_steps` steps old and

$$\frac{\|\bar{C}_t - \bar{C}_{t-k}\|_F}{\|\bar{C}_{t-k}\|_F + \varepsilon_{\text{metric}}} \leq \texttt{eigen_change_tol},$$

the cached decomposition is reused. Otherwise, the eigendecomposition is recomputed and the cache age is reset. The staleness check costs $O(d_{\text{in}}^2)$, which is dominated by covariance formation whenever m is nontrivial. We also cache the power vectors $\lambda_i^{\tau/2}$ and $\lambda_i^{-\tau/2}$ for the current value of τ; these are invalidated whenever the eigendecomposition is refreshed.

Foreach update path. With `torch_foreach=true`, weight decay and parameter updates are applied using PyTorch multi-tensor kernels. Parameters are grouped by their shape-corrected learning rate $\eta\, r(d_{\text{in}}, d_{\text{out}})$, and all tensors in the same group are updated in a single foreach call. This does not fuse the graph-aware metric computation itself, but it reduces Python overhead and GPU kernel-launch latency in the final update stage.

Per-step cost. Table 10 summarizes the per-layer optimizer cost for the full-covariance path with active eigen caching. The Newton–Schulz orthogonalization has the same asymptotic cost as in vanilla Muon. The extra GraphMuon-specific terms are covariance estimation, occasional eigendecomposition, metric application, and graph-aware step normalization. These terms are controlled by node subsampling, the full/diagonal covariance switch, and eigendecomposition caching.

Scalability with covariance sampling. GraphMuon reuses the propagated representations computed during the forward pass, avoiding additional graph propagation. For a covariance sampling budget m and input dimension d, covariance construction requires $O(md^2)$ operations, while cached eigendecomposition adds an amortized $O(d^3/r)$ cost for a refresh interval of r steps. For dimensions above the full-metric threshold, the diagonal approximation avoids the cubic eigendecomposition cost.

Table 11 evaluates the effect of increasing m eightfold, from 512 to 4,096 rows. Optimization remains stable, reaching the fixed validation-loss target within 145–153 epochs across all budgets. Best

Table 10: Asymptotic per-layer optimizer cost for the full-covariance GraphMuon path. Here m is the number of sampled nodes, $d = d_{\text{in}}$, and k is the number of Newton–Schulz iterations.

Operation	Cost	Notes
Read cached H_{prop}	$O(1)$	Tensor already computed in the forward pass
Node subsampling	$O(N)$	Due to `torch.randperm`
Covariance C_g	$O(md^2)$	Gram matrix of sampled propagated features
Staleness check	$O(d^2)$	Relative Frobenius change of $\bar{C}$
Eigendecomposition	$O(d^3)$	Amortized by caching
Power scales $\lambda^{\pm\tau/2}$	$O(d)$	Cached for fixed τ
Metric application	$O(d^2 d_{\text{out}})$	Two applications of $Q(s \odot Q^\top M)$
Newton–Schulz	$O(kdd_{\text{out}} \min(d, d_{\text{out}}))$	Same order as vanilla Muon
Step normalization	$O(d^2 d_{\text{out}})$	Computes $\text{tr}(O^\top C_g O)$
Parameter update	$O(dd_{\text{out}})$	Foreach-batched when enabled

Table 11: Scalability with covariance sampling budget on OGBN-ARXIV. We vary the number of sampled rows m used to estimate the graph-induced covariance. Direction cosine and graph-energy error are measured relative to the full-covariance reference.

Rows m	**Best val. loss ↓**	**Epochs to target ↓**	**Test acc. ↑**	**Direction cosine ↑**	**Graph-energy error ↓**
512	0.8591	153	0.7187	0.9996	0.0098
1,024	0.8599	151	**0.7203**	0.9998	0.0045
2,048	0.8592	148	0.7196	0.9999	0.0024
4,096	0.8598	**145**	0.7198	**1.0000**	**0.0012**

validation loss varies only between 0.8591 and 0.8599, while test accuracy remains between 0.7187 and 0.7203. Larger sampling budgets improve approximation fidelity: direction cosine increases from 0.9996 to 1.0000, while graph-energy error decreases from 0.0098 to 0.0012. These results indicate that accurate graph-aware updates can be obtained using relatively small covariance samples.

The large-scale memory profile in Table 12 provides complementary evidence on OGBN-PRODUCTS. GraphMuon adds approximately 0.005 GB of optimizer-owned memory relative to Muon, without increasing peak CUDA memory, and incurs only 2.8% additional per-step runtime (1.411 versus 1.372 seconds). Thus, covariance sampling and eigendecomposition caching keep the additional graph-aware optimization overhead modest in the evaluated settings.

G.5 Hardware, Software, and Baseline Implementations

All experiments were run on a single NVIDIA RTX 6000 GPU. GraphMuon was implemented in PyTorch and integrated into a shared training pipeline used by all optimizers. Unless stated otherwise, all methods use the same model architectures, dataset splits, input features, batching scheme, evaluation protocol, and hyperparameter tuning budget.

For baselines, AdamW is the standard PyTorch implementation (`torch.optim.AdamW`). Muon and GraphMuon are implemented directly in PyTorch within our training code. Shampoo is implemented using the `pytorch-optimizer` library. For any remaining optimizer baselines, we use the official or original reference implementations when available, adapting only the optimizer interface required for compatibility with our training code. These adaptations do not change the mathematical update rules of the baselines.

Runtime is measured on the same hardware for all methods. In time-to-target experiments, wall-clock time is measured until the validation target is first reached. One-time dataset preprocessing is excluded from the timing. We synchronize CUDA before recording timed intervals to avoid artifacts from asynchronous GPU execution. All reported averages and standard deviations are computed over the same set of random seeds.

Table 12: Memory footprint and per-step runtime on an `ogbn-products`-scale input configuration with approximately 1.2M nodes and hidden dimension 512. GraphMuon increases optimizer-owned memory because it stores a small graph-metric cache, but the absolute overhead is only a few megabytes. Persistent CUDA memory is nearly unchanged, and peak CUDA memory is identical across the profiled optimizers.

Optimizer	State	Graph cache	Owned	Persistent	Peak CUDA	Step time (s)
AdamW	0.0025	0.000	0.0025	15.807	28.211	1.076
Muon	0.0013	0.000	0.0013	15.806	28.211	1.372
GraphMuon	0.0013	0.005	0.0063	15.811	28.211	1.411

Software environment. All experiments were conducted with up-to-date stable versions of Python, PyTorch [23] (offered under BSD-3 Clause license), CUDA, PyTorch Geometric [8] (offered under MIT license), OGB, and `pytorch-optimizer` as of May 2026.

G.6 Memory Footprint

We profile the memory footprint of GraphMuon relative to AdamW and vanilla Muon on an `ogbn-products`-scale input configuration with approximately 1.2M nodes and hidden dimension 512. Table 12 reports optimizer state memory, GraphMuon-specific graph-cache memory, optimizer-owned memory, persistent CUDA memory, peak CUDA memory, and average step time. Optimizer-owned memory includes both optimizer state and any GraphMuon-specific cached metric quantities. Persistent CUDA memory measures the memory that remains allocated during training, while peak CUDA memory reports the maximum CUDA allocation observed during the profiled step.

GraphMuon adds 0.0050 GB of graph-cache memory in this profiling run, bringing its total optimizer-owned memory to 0.0063 GB. Although this is larger than AdamW in relative terms, the absolute increase is small compared to the total training memory. Persistent CUDA memory increases only from 15.8077 GB for AdamW to 15.8115 GB for GraphMuon, corresponding to a negligible end-to-end memory increase. Peak CUDA memory is identical across all three optimizers at 28.2111 GB.

GraphMuon has a slightly higher per-step time than vanilla Muon in this setting (1.411s vs. 1.372s), reflecting the additional covariance estimation, metric application, and graph-aware normalization. This overhead is small in absolute terms and does not materially increase peak GPU memory usage.

G.7 Selected GraphMuon Configurations

Table 13 reports the learning rate and graph-awareness strength τ used for the selected GraphMuon configurations. For each dataset, GraphMuon denotes the configuration selected by best validation accuracy, while GraphMuon* denotes the fastest configuration that reaches AdamW's validation target. If no GraphMuon* row is shown, the two selection criteria choose the same configuration.

Table 13: Selected learning rate and graph-awareness strength τ for GraphMuon. GraphMuon denotes the validation-selected configuration, while GraphMuon* denotes the fastest configuration reaching AdamW's validation target.

Dataset	Optimizer	Learning rate	τ
ogbn-arxiv	GraphMuon	0.01	0.1
ogbn-mag	GraphMuon	0.10	0.5
	GraphMuon*	0.15	0.5
ogbn-products	GraphMuon	0.01	0.1
	GraphMuon*	0.03	0.3
questions	GraphMuon	0.03	0.5
	GraphMuon*	0.03	0.3

Table 14: Complete validation-loss time-to-target results across 15 dataset–architecture configurations. Each optimizer entry reports epochs / wall-clock seconds. Speedups are measured against AdamW's wall-clock time; **bold** indicates configurations where GraphMuon achieves the lowest time-to-target. – denotes failure to reach the target within the training budget.

Dataset	Arch.	Target	AdamW (ep/s)	Muon (ep/s)	Muon ×	GraphMuon (ep/s)	GraphMuon ×
ogbn-arxiv	GAT	1.0118	122 / 1478.8	–	–	**12 / 161.2**	**9.17**
ogbn-arxiv	GCN	0.8753	433 / 30.3	188 / 16.6	1.83	**164 / 13.1**	**2.32**
ogbn-arxiv	GraphGPS	1.7066	24 / 1345.2	2 / 121.1	11.11	**1 / 83.7**	**16.07**
ogbn-mag	GAT	2.7167	69 / 1857.4	6 / 160.4	11.58	**3 / 110.2**	**16.85**
ogbn-mag	GCN	2.6930	100 / 58.3	65 / 38.1	1.53	**41 / 24.1**	**2.42**
ogbn-mag	GraphGPS	3.0715	21 / 1978.3	3 / 406.3	4.87	**1 / 223.9**	**8.84**
ogbn-products	GCN	0.3182	300 / 1663.5	–	–	**226 / 1257.3**	**1.32**
cora	GCN	0.6873	18 / 0.0922	313 / 1.8	0.05	**10 / 0.0807**	**1.14**
questions	GAT	0.1217	481 / 15.2	320 / 10.2	1.49	**167 / 7.1**	**2.14**
ogbg-molbace	GINE	0.6212	65 / 59.1	–	–	**22 / 29.8**	**1.98**
ogbg-moltox21	GINE	0.2946	9 / 22.3	6 / 22.6	0.99	**4 / 18.6**	**1.20**
ZINC	GINE	0.2502	293 / 946.6	242 / 1038.4	0.91	**106 / 679.0**	**1.39**
ZINC	GraphGPS	0.1584	179 / 3176.7	78 / 2062.1	1.54	**34 / 1192.1**	**2.66**
citeseer	GCN	1.0150	14 / 0.0738	24 / 0.2	0.37	11 / 0.0904	0.82
ZINC	GraphGCN	0.2921	490 / 1487.8	–	–	458 / 1566.1	0.95
Mean GraphMuon speedup (15 configurations)							**4.62**

G.8 Additional Architecture and Task Results

We extend the time-to-target evaluation across 15 configurations, covering GCN, GAT, GINE, GraphGPS, and GraphGCN, as well as node classification, graph classification, and graph regression. For each configuration, the target is the minimum mean validation loss attained by AdamW. We report the first epoch and corresponding cumulative wall-clock time at which each optimizer reaches this target.

Across all 15 configurations, GraphMuon reaches AdamW's validation-loss target in fewer epochs, and achieves the lowest wall-clock time-to-target in 13 of 15 configurations. It reaches the target in all 15 configurations, compared with 11 of 15 for Muon. The arithmetic-mean wall-clock speedup over AdamW is $4.62\times$ across all configurations. The strongest improvements include GAT on OGBN-MAG ($16.85\times$) and GraphGPS on OGBN-ARXIV ($16.07\times$).

The gains are not universal in wall-clock time. On Citeseer with GCN and ZINC with GraphGCN, GraphMuon requires fewer epochs but is slightly slower than AdamW in cumulative time. These cases illustrate that improved epoch-level optimization does not always translate into wall-clock acceleration, particularly when optimizer overhead is substantial relative to the model's training cost.

Table 15: Time-to-target comparison. The target is AdamW's best validation accuracy on each dataset. Results are mean ± standard deviation across seeds. Speedup is AdamW mean wall-clock time divided by optimizer mean time-to-target. Unmarked rows are selected by best validation accuracy; * marks the fastest configuration reaching AdamW's target. If no * appears, the two coincide. "–" indicates failure to reach the target within budget.

Dataset	Optimizer	Test Acc.	Epochs to Target	Walltime (s)	Speedup vs. AdamW
ogbn-arxiv	AdamW	0.7222 ± 0.0022	433.4 ± 59.3	30.427 ± 4.010	$1.00\times$
	Muon	0.7197 ± 0.0017	284.5 ± 26.8	19.734 ± 2.176	$1.54\times$
	Muon*	0.7200 ± 0.0010	198.4 ± 54.3	13.442 ± 3.677	$2.26\times$
	Natural Gradient	0.7194 ± 0.0005	264.5 ± 15.6	17.563 ± 1.106	$1.73\times$
	Shampoo	0.7177 ± 0.0027	–	–	–
	GraphMuon	0.7196 ± 0.0009	130.2 ± 8.8	10.171 ± 0.781	$2.99\times$
ogbn-mag	AdamW	0.3267 ± 0.0031	90.1 ± 9.2	52.576 ± 5.398	$1.00\times$
	Muon	0.3441 ± 0.0081	43.6 ± 6.8	25.587 ± 3.977	$2.05\times$
	Muon*	0.3426 ± 0.0051	42.7 ± 6.4	25.068 ± 3.763	$2.10\times$
	Natural Gradient	0.3457 ± 0.0019	35.8 ± 2.0	20.986 ± 1.172	$2.51\times$
	Shampoo	0.1632 ± 0.0108	–	–	–
	GraphMuon (Ours)	0.3590 ± 0.0048	25.8 ± 1.4	15.207 ± 0.825	$3.46\times$
	GraphMuon* (Ours)	0.3532 ± 0.0093	18.6 ± 2.3	11.067 ± 1.359	$4.75\times$
ogbn-products	AdamW	0.7455 ± 0.0029	266.0 ± 28.7	1474.944 ± 159.170	$1.00\times$
	Muon	0.7462 ± 0.0031	165.4 ± 9.0	915.521 ± 49.218	$1.61\times$
	Muon*	0.7417 ± 0.0031	148.7 ± 13.7	824.160 ± 75.388	$1.79\times$
	Natural Gradient	0.7462 ± 0.0027	176.0 ± 26.4	979.136 ± 146.911	$1.51\times$
	Shampoo	0.6004 ± 0.0131	–	–	–
	GraphMuon (Ours)	0.7598 ± 0.0032	209.5 ± 7.3	1165.575 ± 40.489	$1.27\times$
	GraphMuon* (Ours)	0.7458 ± 0.0043	124.8 ± 8.2	694.053 ± 45.522	$2.13\times$
questions	AdamW	0.9707 ± 0.0004	226.0 ± 135.8	2.834 ± 1.649	$1.00\times$
	Muon	0.9708 ± 0.0002	94.3 ± 7.7	1.256 ± 0.106	$2.26\times$
	Muon*	0.9702 ± 0.0005	55.0 ± 0.0	0.747 ± 0.000	$3.79\times$
	Natural Gradient	0.9707 ± 0.0003	35.5 ± 9.2	0.463 ± 0.122	$6.12\times$
	Shampoo	0.9709 ± 0.0001	72.5 ± 13.4	6.317 ± 1.260	$0.45\times$
	GraphMuon (Ours)	0.9709 ± 0.0002	14.8 ± 3.6	0.436 ± 0.106	$6.50\times$
	GraphMuon* (Ours)	0.9709 ± 0.0002	13.0 ± 0.8	0.384 ± 0.025	$7.38\times$

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state the main contribution: GraphMuon adapts Muon-style matrix updates to the graph-propagated representation geometry of GNN layers. The theoretical and empirical claims are supported in Sections 4–5, with the empirical scope limited to the graph-learning benchmarks evaluated in the paper.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper discusses limitations, including regimes where the graph-induced covariance may be noisy or low-rank, especially on small inductive graph-level tasks.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: The theoretical results state their assumptions explicitly. Complete proofs for the perturbation-control result, sharpness proposition, and graph-metric Procrustes result are provided in Appendix E.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The paper provides the full algorithm, datasets, evaluation protocol, baselines, hyperparameter-selection procedure, and selected GraphMuon configurations. Additional implementation and environment details are provided in Appendices C and G.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [No]

   Justification:The experiments use publicly available benchmark datasets. We will release the GraphMuon code, including full experimental scripts and usage instructions, to support reproducibility and future work.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: Section 5 describes the datasets, backbones, optimizer baselines, evaluation protocol, and time-to-target metric. Appendix G provides dataset statistics, architectural details, optimizer grids, implementation details, and selected hyperparameters.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

Justification: The main experimental tables report means and standard deviations across random seeds. The paper states that the same set of random seeds is used across optimizers and that wall-clock measurements are averaged over these runs.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper reports wall-clock time-to-target results and includes a memory-footprint analysis comparing optimizer state, graph-cache memory, persistent CUDA memory, peak CUDA memory, and per-step time. Additional software and hardware environment details are provided in Appendix G.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research uses public benchmark datasets and does not involve human-subject experiments, private data, or high-risk model release. The submission is anonymized and follows the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.

- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: This work is foundational research on optimization for graph neural networks and is not tied to a specific deployment domain. Its primary expected impact is improved training efficiency for GNNs.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
    - The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
    - The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
    - If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper does not release high-risk models, scraped datasets, generative models, or systems with an obvious dual-use misuse risk. Therefore, additional release safeguards are not applicable.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: The paper uses public benchmark datasets and standard open-source software packages, which are cited in the paper.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: We don't release any new assets.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing nor research with human subjects. Guidelines

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.

- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: We don't involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
    - We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
    - For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: We used only for writing, editing, and drafting assistance, which does not affect the scientific methodology or results.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.