Efficient End-to-end Language Model Fine-tuning on Graphs

Rui Xue
rxue@ncsu.edu
North Carolina State University, Raleigh, USA

Xipeng Shen
North Carolina State University, Raleigh, USA

Ruozhou Yu
North Carolina State University, Raleigh, USA

Xiaorui Liu
North Carolina State University, Raleigh, USA

Abstract

Learning from Text-Attributed Graphs (TAGs) has attracted significant attention due to its wide range of real-world applications. The rapid evolution of language models (LMs) has revolutionized the way we process textual data, which indicates a strong potential to replace shallow text embedding generally used in Graph Neural Networks (GNNs). However, we find that existing LM approaches that exploit text information in graphs suffer from inferior computation and data efficiency. In this study, we introduce *LEADING*, a novel and efficient approach for end-to-end fine-tuning of language models on TAGs. To enhance data efficiency, *LEADING* efficiently transfers rich knowledge from LMs to downstream graph learning tasks with limited labeled data by employing end-to-end training of LMs and GNNs in a semi-supervised learning setting. To address associated computation efficiency issues, it introduces two techniques: neighbor decoupling targeting LMs and implicit graph modeling targeting GNNs, respectively. Our proposed approach demonstrates superior performance, achieving *state-of-the-art (SOTA)* results on the ogbn-arxiv leaderboard, while maintaining computation cost and memory overhead comparable to graph-less fine-tuning of LMs. Through comprehensive experiments, we showcase its superior computation and data efficiency, presenting a promising solution for various LMs and graph learning tasks on TAGs. The code is available at https://github.com/RXPHD/LEADING.

CCS Concepts

• **Computing methodologies** → **Neural networks**; **Supervised learning by classification**; **Information extraction**.

Keywords

Graph Neural Networks, Language Models, End-to-end Training

ACM Reference Format:
Rui Xue, Xipeng Shen, Ruozhou Yu, and Xiaorui Liu. 2025. Efficient End-to-end Language Model Fine-tuning on Graphs. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2 (KDD '25), August 3–7, 2025, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3711896.3736922

KDD '25, Toronto, ON, Canada

ACM ISBN 979-8-4007-1454-2/2025/08
https://doi.org/10.1145/3711896.3736922

1 Introduction

Graph neural networks (GNNs) have been widely used for representation learning on graph-structured data [14], and they achieve promising state-of-the-art performance on various graph learning tasks, such as node classification, link prediction, and graph classification. Numerous graphs in real-world applications exhibit nodes that are associated with textual attributes, leading to the prevalence of text-attributed graphs (TAGs). TAGs provide a graph-based framework for representing textual data and the connections between them through edges. The fusion of textual attributes and graph topology constitutes valuable information, bolstering representation learning in real-world applications such as recommender systems [20], citation graphs [18], social networks [13], and knowledge graphs [36].

In the context of GNNs, shallow text embeddings such as Bag-of-Words [15] and Word2Vec [29] are usually extracted from raw textual data and used as the numerical node attributes in GNNs due to their superior simplicity and efficiency. However, as they do not fully capture the complex textual semantic features, these approaches inherently restrict the performance of downstream tasks. On the other hand, the recursive feature aggregation in GNNs results in the well-known neighborhood explosion problem [13] such that the computation of each node involves its L-hop neighbors with L feature aggregation layers. This not only leads to significant scalability challenges but also limits the exploration of more complex and powerful deep learning techniques such as LMs for the representation learning on TAGs.

Recently, researchers have begun to explore the potential of pre-trained language models (LMs), such as BERT [7], DeBERTa [16] and DistilBERT [32], for representation learning on TAGs due to their unprecedented capabilities in language understanding and generation across a wide range of tasks. The commonly adopted approach follows a *cascaded structure* [5]. This entails an initial LM fine-tuning step on downstream tasks such as node classification. Subsequently, the text embeddings extracted from the fine-tuned LMs are leveraged as the initial node features for downstream GNNs. Although the cascaded structure is efficient, graph structural information is not incorporated in the fine-tuning of LMs, resulting in sub-optimal performance. To address this issue, the *iterative structure* has also been explored for the joint training of LMs and GNNs. For instance, GLEM [41] trains LMs and GNNs separately in an iterative manner by generating pseudo labels for each other. In addition, *self-supervised learning* has also been proposed to enhance LMs fine-tuning by link prediction tasks, exemplified by GIANT [6].

The aforementioned works demonstrate the potential of exploiting LMs on TAGs. However, these approaches still face limitations in *data efficiency* or *computation efficiency*. First, both cascaded and

iterative structures encounter significant data inefficiency. When the labeled data is scarce, these methods struggle to effectively transfer the required knowledge for downstream tasks as the fine-tuning strategies do not utilize labeled data efficiently. Second, both iterative structures and the self-supervised learning approach introduce a substantial increase in computational overhead. This elevated computational cost poses significant scalability challenges, especially when dealing with large-scale datasets. These shortcomings tremendously limit their applications in transferring the rich knowledge of LMs to facilitate representation learning on TAGs.

In this paper, our aim is to develop an efficient algorithm for fine-tuning LMs that not only effectively adapts LMs to downstream tasks with limited labeled data (*data efficiency*) but also demonstrates superior scalability (*computation efficiency*). We argue that end-to-end training of LMs and GNNs is crucial for achieving data efficiency, as it enables superior knowledge fusion between the two, leveraging the unique advantages of GNN message passing techniques. However, it faces challenges due to scalability (computation efficiency) issues, which we attribute to the giant size of LMs used and the neighbor explosion issue in GNNs. To tackle this, we identify *computation redundancy* as the bottleneck hindering end-to-end training. We further decompose this issue into *encoding redundancy* in LMs and *propagation redundancy* in GNNs. To address these problems, we propose *neighbor decoupling* and *implicit graph modeling* as solutions to alleviate these two redundancy issues respectively. Finally, with the aid of the proposed techniques, we make end-to-end training of LMs and GNNs feasible. Our algorithm demonstrates superior performance, achieving state-of-the-art results on ogbn-arxiv, and exhibits strong scalability comparable to graph-less LM fine-tuning. Therefore, it offers a promising solution for a wide range of LMs and graph learning tasks on TAGs. Moreover, our novel design addresses the neighbor explosion problem at its source and reduces the staleness issue inherent in existing cache-based methods, thereby offering significant advantages for large-scale GNN training.

2 Methodology

GNNs have been proven to be data-efficient due to their excellent prediction performance on semi-supervised learning where only very limited labeled data is available. The data efficiency of GNNs can be largely attributed to their ability to integrate node attributes and graph structure information in a unified message-passing framework. Through end-to-end training, it leverages the scarce labeled data to provide informative supervision for the vast pool of unlabeled nodes. However, GNNs' data efficiency comes with the sacrifice of computation efficiency [13]. Hence, most existing approaches exploiting LMs for learning on TAGs fall short in data efficiency and thus fail to effectively adapt the rich knowledge in LMs to downstream graph learning tasks as discussed in Section 1 and Section 4. We hypothesize that their data inefficiency originates from the fact that existing methods can not fine-tune LMs with graph learning in an end-to-end manner due to the scalability challenges in both LMs and GNNs.

Building upon the analyses presented above, conducting end-to-end training emerges as a critical factor for enhancing the transfer of knowledge from LMs to the specific downstream tasks. However, the primary challenge we must address is the accompanying scalability (computation efficiency) issue that hinders the application of end-to-end training. We identify the primary challenge as computation redundancy. Consequently, we propose a novel end-to-end fine-tuning strategy (LEADING) to alleviate these redundancies, leading to a highly efficient and scalable solution. We also present a theoretical proof in Appendix C showing that LEADING can better facilitate knowledge transfer via end-to-end training. Before that, we first introduce the notations as follows.

Notations. A graph is represented by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{v_1, \ldots, v_N\}$ is the set of nodes and $\mathcal{E} = \{e_1, \ldots, e_M\}$ is the set of edges. For a text-attributed graph, each node v_i is associated with a sequential of raw text feature. We denote the d-dimensional hidden feature vectors of nodes as $\mathbf{X} \in \mathbb{R}^{N \times d}$. The graph structure of $\mathcal{G}$ can be represented by an adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, where $\mathbf{A}_{ij} > 0$ when there exists an edge between node v_i and v_j, and $\mathbf{A}_{i,j} = 0$ otherwise. The symmetrically normalized graph Laplacian matrix is defined as $\tilde{\mathbf{L}} = \mathbf{I} - \tilde{\mathbf{A}}$ with $\tilde{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ where $\mathbf{D}$ is the degree matrix.

2.1 Computation Redundancy in LM-GNN

Various sampling approaches have been proposed to enhance the scalability and efficiency of GNN training. However, integrating LMs with GNNs in an end-to-end training paradigm introduces its own unique hurdles. This is primarily due to the substantial computational and memory costs associated with LMs, given their large sizes. Additionally, the employment of deep GNNs to capture long-range dependencies poses neighbor explosion challenge to end-to-end LM+GNN training. To tackle the challenges present in both LMs and GNNs, we begin by offering a novel and insightful analysis of computation redundancy within the end-to-end training framework. We identify this redundancy as a crucial factor in achieving successful end-to-end training. We classify the redundancy into encoding redundancy in LMs and propagation redundancy in GNNs. Subsequently, we propose two corresponding techniques, namely neighbor decoupling and implicit graph modeling, to address these issues.

Encoding Redundancy in LMs. In the integration of LMs with GNNs, we have to adopt mini-batch sampling to reduce the computation and memory costs. However, existing sampling strategies of GNNs exhibit heavy redundancy that requires frequently repeated LM encoding of node features, which becomes especially significant due to the immense size of these LMs. Taking the mini-batch sampling in Figure 1 as an example, the node features need to be encoded by LMs multiple times through every epoch, either as target nodes themselves or as neighbors of other target nodes. For example, V_1 serves as a target node in Batch 1 and serves as a neighbor node in Batch 2 and Batch 3. However, the LM embedding of the node features will not have notable changes between the mini-batch iterations due to the nature of model fine-tuning.

The above analysis implies that a significant amount of computation on LM encoding is redundant. This redundancy becomes particularly considerable when we employ smaller batch sizes, as typically used in LMs, as well as when we introduce more aggregation layers to capture long-distance information in GNNs. According to our statistical analysis on ogbn-arxiv dataset, during each

epoch in the training of a 2-layer GNN with neighbor sampling, the node feature of each node is encoded as a target node only once but as a neighbor node 19 times on average when the batch size is 1024 (25 times when the batch size is 64). For a 5-layer GNN that requires sampling from 5-hop neighbors, the node feature of each node is encoded 96 times as a neighbor node on average. This statistical analysis verifies the LM encoding redundancy in mini-batch GNNs.

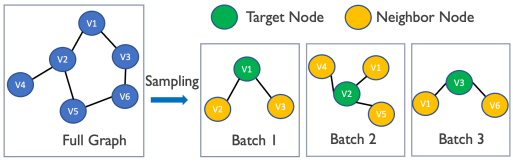

Figure 1: Encoding Redundancy in Mini-batch GNNs.

Propagation Redundancy in GNNs. Besides the LM encoding redundancy, there also exists propagation redundancy in GNNs. As discovered by a recent work [37], the node embedding in the GNN layers will not change notably over the training iterations but the node information is propagated multiple times repeatedly in each iteration to capture long-distance dependency on graphs. This propagation redundancy causes huge sampling, memory, and computation costs that increase significantly with the number of aggregation layers employed. Next, we will propose a Language models fine-tuning on Graphs (LEADING) algorithm that tackles the encoding redundancy in Section 2.2 and propagation redundancy in Section 2.3.

2.2 LEADING in LMs: Neighbor Decoupling

Due to the huge computation and memory cost of LMs, it is imperative to reduce the redundant LM encoding computation. Our first observation is that for a sampled subgraph, only the target nodes obtain accurate aggregated features and gradients. On the contrary, the major role of neighbor nodes is to facilitate predictions for target nodes but they may not obtain accurate aggregation features and gradients due to their missing neighbors. In other words, the mini-batch neighbor sampling tries to maximally maintain the neighbors of target nodes but the neighbors of neighbors might be out of the batch. As a result, it is feasible to only use the gradient of target nodes to update the LMs. The second key observation is that the LM embedding does not change rapidly during the fine-tuning stage, indicating that it is unnecessary to update the LM embedding of neighboring nodes in real-time, considering the significant computational resources that the LM requires.

Neighbor Decoupling. These key observations motivate us to design a novel training algorithm that fully decouples the LM computation of target nodes and their neighbor node as shown in Figure 2 and Algorithm 1. To reduce the encoding redundancy, we opt to segregate the encoding of target and neighbor nodes into two distinct pipelines. First, for pipeline 2, the LM randomly samples a mini-batch of node text features $\mathbf{T}_2$ from the whole graph and computes their LM embedding $\mathbf{X}_2$ without requiring gradient (line 5). It then caches into memory (line 6), which helps facilitate the rapid filling and refreshing of the cache for use in the first pipeline. Clearly, the memory will at least be filled after the first epoch if $\mathbf{X}_1$ and $\mathbf{X}_2$ have the same shape. Following this, neighbor sampling takes place in pipeline 1, where the LM computes the encoding solely for the target nodes $\mathbf{T}_1$ (line 7) within the current mini-batch.

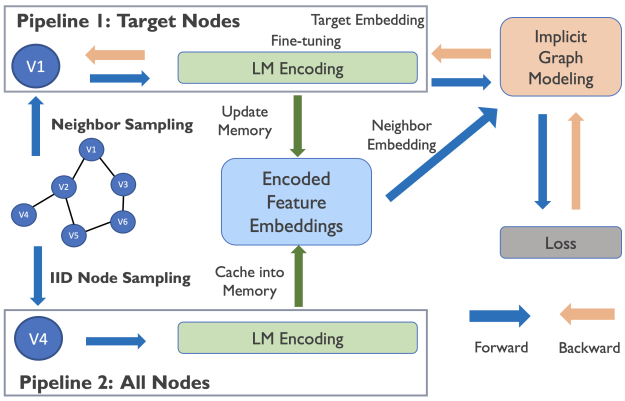

Figure 2: LEADING: two-pipeline training process. (a) a randomly sampled batch is encoded in pipeline 2 and is stored in memory. (b) only the target nodes within the neighbor-sampled batch are encoded with gradients in pipeline 1. (c) Neighbor nodes' embeddings are retrieved and the resulting subgraph is then fed into GNNs. (d) Gradients from target nodes are employed to fine-tune the language models.

Since only the target nodes require gradients, it can significantly reduce memory costs. Similarly, the computed LM embedding $\mathbf{X}_1$ from pipeline 1 (line 8) will be cached in the memory to update the embedding and reduce staleness. Next, the encoded embeddings of neighboring nodes $\mathbf{X}_{\text{neighbor}}$ are retrieved from the memory bank according to their indexes (line 9) and concatenated with $\mathbf{X}_1$ to form the node embedding for the entire subgraph (line 10) before being fed into GNNs (line 11). Finally, after computing the loss, backward propagation occurs exclusively within the first pipeline to fine-tune the LMs, as gradients are only required for target nodes.

Algorithm 1 LEADING Algorithm—Neighbor Decoupling

1: **Input:** Input Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, Pre-trained LM $f(\mathbf{T}, \Theta^0)$ State
Output: Fine-tuned LM $f(\mathbf{T}, \Theta^*)$
2: **Begin**
3: **for** each mini-batch text $\mathbf{T}_1$ in pipeline 1;
4: each mini-batch text $\mathbf{T}_2$ in pipeline 2 **do**
5: $\mathbf{X}_2 = f(\mathbf{T}_2, \Theta^k)$: Neighbor Nodes Encoding (without gradients)
6: Cache into Memory $\mathbf{M} \leftarrow \mathbf{X}_2$
7: $\mathbf{X}_1 = f(\mathbf{T}_1[\text{target}], \Theta^k)$: Target Nodes Encoding (with gradients)
8: Cache into Memory $\mathbf{M} \leftarrow \mathbf{X}_1$
9: Retrieve Neighbor Embeddings $\mathbf{X}_{\text{neighbor}} \leftarrow \mathbf{M}$
10: $\mathbf{X}_{\text{in}} = \text{Concat}(\mathbf{X}_1, \mathbf{X}_{\text{neighbor}})$
11: $\mathbf{X}_{\text{out}} = \text{GNN}(\mathbf{X}_{\text{in}})$
12: Compute Loss and Gradient Update
13: **end for**

This two-pipeline neighbor decoupling approach offers two significant benefits. First, it completely resolves the encoding redundancy problem, as the encoding times of each node in each epoch are controlled by the batch size used in the second pipeline. For instance, if two pipelines take the same batch size, each node feature only needs to be encoded by LM twice, which significantly reduces the computation cost. Second and more importantly, the

memory cost is significantly reduced because the back-propagation training pipeline only processes target nodes. This eliminates the need to depend on a large number of neighbor nodes—a clear advantage over existing caching methods that still rely on them (see Section 3.3). Consequently, by decoupling the majority of nodes in the mini-batch from the neighbor nodes, we **fundamentally resolve the neighbor explosion problem at its root.** This is not only the key to achieving LM+GNN end-to-end fine-tuning, but also a significant contribution to solving the neighbor explosion problem in large-scale GNN training—a feat that has never been accomplished by other models. We also theoretically demonstrate in Appendix D that it achieves variance reduction.

2.3 LEADING in GNNs: Implicit Graph Modeling

While the proposed neighbor decoupling approach significantly reduces memory costs, it is important to note that deep GNN models are typically employed to capture long-distance dependencies in graphs. This introduces a substantial memory overhead, as each intermediate feature embeddings have to be stored to facilitate gradient computation. Motivated by recent advances in implicit models such as Neural ODE [4], IGNN [12], DEQ [1] as well as the unified view of graph signal denoising [26], forward propagation can be efficiently computed through a fixed point solver:

$$\mathbf{X}_* = \text{Solver}(f_\theta(\mathbf{X}_*, \mathbf{X}_{in})), \lim_{l\to\infty} \mathbf{X}_l = \mathbf{X}_* \tag{1}$$

where $f_\theta(\mathbf{X}_*, \mathbf{X}_{in}) = \mathbf{X}_*$ is the implicit function, $\mathbf{X}_*$ represents the fixed point of node embeddings, which corresponds to the final values of the hidden layer in a network when the number of layers tends to infinity $l \to \infty$. This implicit modeling offers two main advantages: First, the fixed point $\mathbf{X}_*$ represents the equilibrium state achieved after an infinite number of propagations, with only the fixed point and input embeddings needing to be stored for the backward pass, thereby avoiding the storage of any intermediate values and efficiently capturing long-range information. Second, it provides flexibility for the fixed point solver, as the training and inference of implicit deep learning models are independent of the computational trajectory. In other words, any solver can be used to construct the propagation layers. In this work, we have chosen to incorporate the APPNP [23] for its role as an iterative solver for fixed points, as it has been proven to effectively alleviate oversmoothing in deep GNN models. The theoretical explanation can be found in Appendix B. The fixed-point embeddings precisely match the exact embeddings as outlined below:

$$\mathbf{X}_* = \alpha\left(\mathbf{I} - (1-\alpha)\tilde{\mathbf{A}}\right)^{-1}\mathbf{X}_{in}, \tag{2}$$

$$\mathbf{X}_{l+1} = (1-\alpha)\tilde{\mathbf{A}}\mathbf{X}_l + \alpha\mathbf{X}_{in}, \lim_{l\to\infty} \mathbf{X}_l = \mathbf{X}_* \tag{3}$$

Furthermore, to address propagation redundancy issue, we use $\mathbf{X}_L^{k-1}$ from the previous training iteration $k-1$ as the initial embedding for the current iteration k since features have been propagated over the graph many times in previous iterations and node embeddings in the GNN layers do not change significantly across iterations, as discussed in Section 2.1. In other words, as we know the choice of the initial point affects the convergence, $\mathbf{X}_L^{k-1}$ serves as a more favorable initialization for the fixed point solver due to its proximity to the fixed point and stability across training iterations. By doing this, only a very few feature aggregation layers (i.e. iteration steps) are required to approximate the fixed-point solution $\mathbf{X}_*$ in Eq. (2). Then in Algorithm 1, the forward propagation can be formulated as:

$$\mathbf{X}_{\text{in}}^k = \text{Concat}\{\mathbf{X}_1, \mathbf{X}_{\text{neighbor}}\},\ \mathbf{X}_0^k = \mathbf{X}_L^{k-1}, \tag{4}$$

$$\mathbf{X}_{l+1}^k = (1-\alpha)\tilde{\mathbf{A}}\mathbf{X}_l^k + \alpha\mathbf{X}_{\text{in}}^k,\ \forall l = 0, \ldots, L-1, \tag{5}$$

where l and k denote the index of layers and training iterations, respectively. The backward propagation can be directly calculated based on the fix-point Eq.(2) by taking derivatives and approximating the matrix inversion iteratively:

$$\mathbf{G}_L^k = \mathbf{G}_0^{k-1}, \tag{6}$$

$$\mathbf{G}_l^k = (1-\alpha)\tilde{\mathbf{A}}\mathbf{G}_{l+1}^k + \alpha\frac{\partial\mathcal{L}}{\partial\mathbf{X}_L^k},\ \forall l = L-1, \ldots, 0, \tag{7}$$

where $G_L = \frac{\partial\mathcal{L}}{\partial\mathbf{X}_L} \approx \frac{\partial\mathcal{L}}{\partial\mathbf{X}_*}$. Please refer Appendix A for detailed proof. Similarly, the backward propagation starts from the gradient in previous iterations $\mathbf{G}_0^{k-1}$, which serves as a better initialization. Finally, the gradient of target nodes can be retrieved from $\mathbf{G}_0^k$ and used for further back-propagation in the LM $f(\mathbf{T}_1, \Theta^k)$. After the end-to-end fine-tuning, we can utilize the fine-tuned LM to generate feature embedding, which serves as the initial embedding for any downstream GNNs and graph learning tasks.

2.4 Computation Complexity Analysis

LM Complexity Analysis. Suppose N is the total number of nodes and C is the computation complexity of encoding one node feature by LMs. Additionally, let P stand for the average encoding times per node in LEADING and Q is the average encoding time per node in existing scalable GNNs, such as GraphSAGE or GAS [11]. Then, the total computation complexity of LM encoding is $O(PNC)$ for LEADING and $O(QNC)$ for other baselines. Given that $P \ll Q$ (where P is constant and equals 2 in our experiments), as discussed in Section 2.1 and shown in Figure 5, LEADING evidently achieves better computation efficiency.

Regarding memory complexity, suppose $O(S)$ is the memory complexity for executing forward and backward propagation per node. For the mini-batch sampling, suppose T and B are the batch sizes of target nodes and neighbor nodes, respectively. Typically, we have $B \gg T$. Then the total memory complexity for LEADING is $O(TS)$, which is the same as training LMs without using graphs. It is much lower than the normal GNN training strategy whose memory complexity is $O((B+T)S)$. These complexity analyses indicate the intriguing scalability of LEADING in the LM phase.

GNN Complexity Analysis. Due to the variety of downstream GNN structures used in baselines, which are not the focus of this section, we use the GraphSAGE an an example to simplify the complexity analysis and directly demonstrate the superior efficiency of our model. Suppose N is the total number of nodes, L is the number of propagation layers, and H is the size of hidden units, and R is the number of sampled neighbors of each node. The time complexity of GraphSAGE is $O(R^L NH^2)$ [24]. It is worth noting

that LEADING has fewer layers $\tilde{L} \ll L$ compared to existing approaches as described in Section 2.3. This reduction in the number of layers contributes to lowering the overall computation cost in feature aggregation.

For memory complexity, regular GraphSAGE requires $O(R^L TH)$ to store the intermediate state at each feature aggregation layer. This complexity grows exponentially with the number of layers. However, our algorithm achieves a memory complexity of $O(TH)$ because we utilize implicit gradient modeling, which does not requires the storage of feature in intermediate layers. Therefore, the memory cost is independent of the number of aggregation layers. This indicates a significant reduction in terms of memory cost.

Table 1: Complexity Analysis

Method	Time	Memory
LM+SAGE	$O(QNC + R^L NH^2)$	$O((B+T)S + R^L TH)$
Cascaded	$O(NC + R^L NH^2)$	$O(TS + R^L TH)$
GLEM	$O(K(NC + R^L NH^2))$	$O(TS + R^L TH)$
LEADING	$O(NC + R^{\tilde{L}} NH^2)$	$O(TS + TH)$

K is the number of iterative training rounds of LM and GNN in GLEM; *LM + SAGE* denotes training the language model and GNN jointly.

3 Experiment

In this section, we present experiments to demonstrate the superior data efficiency and computation efficiency of LEADING. In particular, we try to answer the following questions: (Q1) Data efficiency: can our LEADING algorithm transfer the knowledge from LMs to downstream graph learning tasks effectively with limited training data? (Section 3.1, 3.4, 3.5) and (Q2) Computation efficiency: can our LEADING algorithm be more scalable compared with other fine-tuning paradigms and GNNs? (Section 3.2, 3.3)

Datasets. We conduct experiments on both small and large text-attributed graph datasets including Cora [28], PubMed [33], ogbn-arxiv and ogbn-products [18]. We evaluate the effectiveness of LM fine-tuning by taking semi-supervised node classification problems as the downstream tasks. We randomly split the data into training/val/test sets 10 times for Cora and PubMed and report the mean and variance of accuracy following existing works [22]. We adopt the default labeling ratios of these datasets, i.e., 20 training nodes per class for Cora and PubMed (low labeling rate) and 53.7% for ogbn-arxiv (high labeling rate). Additionally, we present a specific case with a high labeling rate on Cora. Please refer to Appendix F for details.

Baselines. We compare the proposed LEADING algorithm with some classic baseline and a set of LM fine-tuning strategies: (1) *Shallow Embedding:* Default shallow embeddings provided by PyG [10]; (2) *Pre-trained LMs:* LMs function as simple encoders without fine-tuning on labeled data, and the resulting feature embeddings are used as inputs for downstream GNNs; (3) *Supervised-FT LMs:* LMs are directly fine-tuned using the labeled data under the supervised setting. Subsequently, the text embedding generated by the fine-tuned LMs is used as the node embedding for downstream GNNs; (4) *GIANT & GLEM:* We choose GIANT [6] and GLEM [41] as the major baselines since they exhibit the excellent performance among all existing works[5]. Moreover, GLEM is a representative method of iterative training strategy, while GIANT is a representative method of self-supervised training strategy. It is worth noting that due to the high training costs of GIANT, we only use pre-trained features provided by their official repository. (5) *GraD & TAPE:* We have incorporated another newly proposed model, GraD[27], which has demonstrated strong performance on the OGB leaderboard and has publicly available code. Additionally, we include TAPE[17], which employs LLMs as an enhancer to generate explanations to aid in boosting downstream performance. We adhere to their experimental settings, including the datasets used (ogbn-arxiv). It's worth noting that TAPE is entirely orthogonal to our algorithm; hence, we also report the performance of LEADING augmented with TAPE features. (6) *Others:* We include several end-to-end joint training paradigms and state-of-the-art baselines in Appendix G.

Evaluation setting. For LMs, in order to ensure a fair comparison with the baselines across different datasets, we include the same LMs as used in their respective studies. Specifically, we employ BERT [7] as utilized in GIANT, DeBERTa [16] as applied in GLEM, and one lightweight variant, DistilBERT [32]. Additionally, to further validate the effectiveness of our algorithm, we conduct additional experiments involving LLMs in Section 3.4.

To evaluate the effectiveness of LMs fine-tuning, we extract the CLS (classification) embedding from the last hidden states of fine-tuned encoder-only LMs as the text embeddings, aligning with the configuration used in the baseline models. For downstream GNNs, we conduct performance comparisons on Cora and Pubmed using two classic GNNs, namely GCN [22] and GAT [35]. In the case of ogbn-arxiv dataset, we employ GCN and Rev-GAT [24]. For the ogbn-products dataset, we opt for GraphSage [13] and GAMLP [40] following existing works [5, 41]. We perform all hyperparameter tuning following baselines.

3.1 Prediction Performance

Major baselines We evaluate the effectiveness of LM fine-tuning by comparing the prediction accuracy on downstream GNNs. From Table 2, we can make the following observations:

- In the low labeling setting (Cora and Pubmed), LEADING outperforms all other LM fine-tuning strategies. Notably, compared with Supervised-FT DeBERTa, LEADING significantly boosts the performance of the DeBERTa from 59.2% to 80.6% for GCN and from 57.4% to 81.4% for GAT on Cora. A similar improvement can be observed on PubMed as well. **We conclude that this improvement stems from successfully implementing end-to-end training.**
- Comparing Pre-trained DeBERTa with Supervised-FT DeBERTa, fine-tuning without an end-to-end manner does not provide significant benefits in the low-labeling setting (Cora and PubMed) due to the scarcity of labeled data. However, it can prove more beneficial as the volume of training data increases (ogbn-arxiv).
- We found that all baselines perform poorly in low labeling rate settings, revealing a gap compared to shallow embeddings. We conclude that when training samples are limited, these methods fail to transfer sufficient pre-trained knowledge from LMs to downstream GNN tasks, which supports our claim in Section 1 and is consistent with existing works [5]. However, LEADING does not suffer from this limitation. Thanks to our proposed end-to-end fine-tuning, it effectively fills the performance gap

Table 2: Prediction accuracy (%) of LM fine-tuning strategies. The best are marked as bold.

Dataset	Cora		Pubmed		Arxiv		Products	
Method	GCN	GAT	GCN	GAT	GCN	Rev-GAT	SAGE	GAMLP
Shallow Embedding	82.0 ± 0.7	82.3 ± 0.7	78.9 ± 1.0	77.7 ± 0.9	71.7	73.6	79.7	83.5
Pre-trained DeBERTa	48.5 ± 1.9	51.0 ± 1.2	62.0 ± 0.1	62.6 ± 0.3	45.7	47.8	62.0	82.4
Supervised-FT BERT	77.3 ± 1.7	78.2 ± 1.4	68.6 ± 1.8	68.6 ± 1.4	73.1	73.8	81.8	79.8
Supervised-FT DistilBERT	79.5 ± 1.5	79.2 ± 1.7	72.8 ± 1.2	72.6 ± 1.1	73.0	73.7	81.5	80.4
Supervised-FT DeBERTa	59.2 ± 1.2	57.4 ± 2.0	62.1 ± 0.1	61.6 ± 0.1	74.7	75.8	82.2	80.7
GIANT (BERT) [6]	—	—	—	—	73.3	75.9	83.1	83.7
GLEM (DeBERTa)*[41]	59.2 ± 1.2	57.4 ± 2.0	62.1 ± 0.1	62.6 ± 0.3	75.9	76.9	83.2	85.1
LEADING (BERT)	80.5 ± 0.4	81.6 ± 0.3	79.1 ± 0.5	79.0 ± 1.0	73.8	74.8	83.8	85.7
LEADING (DistilBERT)	**82.5 ± 0.5**	**83.0 ± 0.5**	79.4 ± 0.4	79.2 ± 0.8	73.5	74.3	83.7	85.3
LEADING (DeBERTa)	80.6 ± 0.3	81.4 ± 0.6	**79.5 ± 0.8**	**79.3 ± 0.6**	**76.1**	**77.6**	**84.1**	**86.5**

* In our investigation, we found that in the low labeling ratio case (such as Cora and PubMed), GLEM achieves its highest accuracy when the ratio of pseudo labels is set to 0. In this unique case, GLEM essentially operates similarly to Pre-trained/Supervised Fine-tuning Language Models, relying solely on truth labels for training. This observation aligns with Chen et al. [5].

observed in baselines and performs the best in all scenarios, highlighting its necessity.

- In the high labeling setting (e.g., ogbn-arxiv), LEADING also achieves strong performance. For DeBERTa, LEADING achieves 76.1% and 77.6% accuracy for GCN and Rev-GAT, which are better than GLEM (75.9% and 76.9%), a model that has proven to be very strong in the high labeling setting [5]. In the case of ogbn-products, LEADING achieves 84.1% for SAGE and 86.5% for GAMLP, outperforming all other baseline methods.
- It should be noted that LEADING achieves these remarkable performances with much better computation efficiency and scalability as will be discussed in Section 3.2. Moreover, it demonstrates strong generalizability by allowing seamless substitution of different LM/LLM architectures while consistently delivering great performance, please refer Section 3.4 for more details.

Table 3: Prediction accuracy (%) comparison with other baselines on Arxiv

Method	GraD[27]	TAPE[17]	LEADING	LEADING + TAPE
Rev-GAT	77.2	77.5	77.6	**78.2**

SOTA performance From Table 3, our approach has achieved best performance on the OGB leaderboard[1] for ogbn-arxiv surpassing all recent baselines without the need for additional LLMs as enhancers or augmentations. Notably, by incorporating augmented features from TAPE[17], we effortlessly achieved the state-of-the-art performance. This underscores LEADING's superior capability in terms of performance and the generality of combining it with other techniques. It validates that end-to-end training can indeed be beneficial, consistent with our analysis in Section 2. Importantly, it pushes the boundaries of graph machine learning on TAGs, emphasizing the necessity of this work.

Table 4: Performance on the ogbn-papers100M dataset.

	GIANT	GLEM	LEADING
Acc(%)	69.3	68.3	**69.8**

Results on extremely large graphs We present results on ogbn-papers100M—the largest dataset with the original raw text attributes available. We compare our approach with GIANT and GLEM using GAMLP as the GNN backbone. We report their results as presented in their original papers. As shown in the table below, our method still achieves the best performance on extremely large TAGs, offering notable effectiveness and scalability advantages.

3.2 Efficiency Analysis

In this section, we investigate the computation efficiency and scalability during the LM fine-tuning stage. We select BERT and DeBERTa as the LM architectures since they are used in the baselines of GIANT and GLEM. The results in Table 5 reveal the following noteworthy observations:

- Notably, the proposed LEADING achieves a memory cost that is nearly identical to Supervised-FT LM training without using graphs, which aligns with our expectations. This is attributed to the proposed neighbor decoupling and implicit graph modeling as introduced in Section 2.
- The iterative training strategy, such as GLEM, and self-supervised training strategy, such as GIANT, exhibit significantly higher memory costs or longer running times compared to the cascaded structure, such as Supervised-FT. Specifically, since all our experiments were conducted using a single GPU, replicating the results of GIANT, which originally used 8 V100 GPUs for training, posed challenges. To replicate their results, we attempted to scale up the batch size by a factor of 8, resulting in out-of-memory (OOM) issues. While reducing the batch size is a possible solution, it significantly prolongs the running time and lowers performance. Therefore, we report N/A in the table. This orders-of-magnitude larger computational overhead can be attributed to the fact that GIANT employs a multi-level fine-tuning approach as discussed in Section 4, which results in a

[1] https://ogb.stanford.edu/docs/leader_nodeprop/

Table 5: Scalability comparison between different LM fine-tuning strategies

Dataset	Arxiv		Products	
Method	Memory(GB)	Running Time(S)	Memory(GB)	Running Time(S)
Supervised-FT BERT	11.5	8400	14.0	19900
Supervised-FT DeBERTa	13.6	12200	25.1	24640
GIANT (BERT) [6]	N/A	N/A	N/A	N/A
GLEM (DeBERTa) [41]	13.6	67634	25.2	137760
LEADING (BERT)	11.7	15241	14.9	35748
LEADING (DeBERTa)	13.9	22226	25.9	44388

In Table 1,2,3, we present a common setting where the same batch size is used for both pipelines. However, this table aims to highlight the efficiency of LEADING by showing the cost associated with achieving best performance by doubling batch size of the second pipeline.

significant increase in computation overhead compared to other training strategies [5].

- Hyperparameters tuning: LEADING does not require additional hyperparameters tuning, which further strengthens its efficiency and simplicity. However, GLEM is more complicated and requires additional hyperparameters tuning such as the ratio of generated pseudo labels, the number of iterations of the EM-Step, etc.
- The running time of LEADING is around 0.8 times higher than that of Supervised-FT, which is reasonable since the two pipelines are run in a sequential manner on the same GPU but it can be easily reduced by parallel computing. The memory cost and running time align well with our complexity analysis in Section 2.4.

3.3 Comparison with Scalable and Caching-Based Models

In this section, we present a comparative analysis between our approach and other scalable GNN backbones—including caching-based models—to highlight the distinctiveness and superiority of our design, considering both performance and efficiency metrics. We choose two widely-used scalable GNNs, namely GraphSAGE [13] and one of the state-of-the-art caching-based algorithm GNNAutoScale [11], as our primary baselines. To expedite computations, we employ DistilBERT [32] paired with a 2-layer APPNP in an end-to-end fashion on Cora, Pubmed and ogbn-arxiv. We use same hyperparameters to ensure a fair comparison.

As shown in Table 6, LEADING can achieve comparable or superior results compared to the coupling method, validating the rationale of LEADING as discussed in Section 2.2. And all three approaches converge at the same speed, as illustrated in Figure 3. Importantly, LEADING stands out as significantly more efficient and is the only model capable of end-to-end training on ogbn-arxiv. This is because all other state-of-the-art caching-based GNN training algorithms, such as GAS and GraphFM, require encoding both target nodes and their first-hop neighbors together. When the graph is large, the first-hop neighbor nodes still dominate, leading to encoding redundancy and out-of-memory issues. More importantly, all caching-based models inevitably suffer from stale historical embeddings [38], which significantly undermines their performance. In contrast, our algorithm **fully eliminates the issue of the neighbor explosion and it's effective in reducing the impact of staleness simply by controlling the batch size of the second pipeline**(see Appendix H). Since no gradients are required for neighbor nodes, this approach leads to significant reductions in memory cost and serves as the key to enabling end-to-end training, resulting in substantial performance improvements. The contribution of this novel design is not limited to LM+GNN architectures.

Table 6: Performance and corresponding memory cost comparison between normal LM-GNN training and LEADING.

Method /Accuracy(%)	Cora	Pubmed	Arxiv
LM + GraphSAGE	82.0 ± 0.6	80.3 ± 0.4	OOM
LM + GAS	81.3 ± 0.3	79.5 ± 0.2	OOM
LEADING	81.9 ± 0.7	80.1 ± 0.4	74.3
Method /Memory(MB)	**Cora**	**Pubmed**	**Arxiv**
LM + GraphSAGE	21021	35070	OOM
LM + GAS	15844	19613	OOM
LEADING	2398	2399	26557

We also present a study on the memory usage on Cora and the average LM encoding times of each node on ogbn-arxiv. We assess the memory usage based on two key factors: (1) different batch sizes while keeping 2 hops neighbors (sampling 10 neighbors for the first hop and 5 for the second hop for each node); (2) varying numbers of hops (sampling 10 neighbors for the first hop and 5 for the following hops for each node) while keeping a fixed batch size.

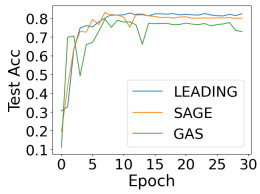

Figure 3: Convergence Curve

The memory usage in Figure 4 reveals that a substantial portion of the computation cost is associated with the encoding of neighboring nodes. Despite GAS achieving improved memory efficiency, especially with a large batch size, and maintaining nearly the same cost as GNNs grow deeper, it still demands significantly more memory than LEADING due to its coupling of 1-hop neighbors. Importantly, LEADING sustains a consistent memory cost comparable to Supervised-FT LMs ("Targets Only") and remains independent of the number of neighbors included, underscoring a notable scalability advantage.

The average LM encoding times of each node in Figure 5 indicate a considerable level of computational redundancy and this

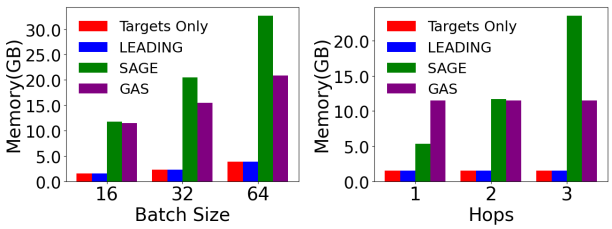

Figure 4: Memory Cost

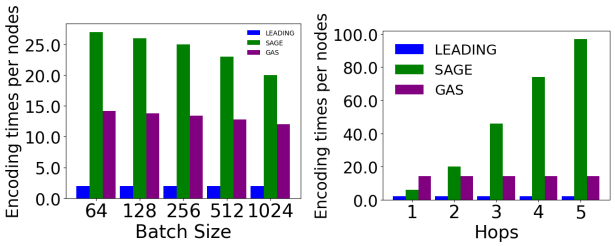

Figure 5: Encoding Times

redundancy is affected by both batch size and number of neighbors. The results also show that our LEADING algorithm ensures that in each epoch, each node is only encoded by LM twice, significantly minimizes computation redundancy due to the decoupled computation of target nodes and neighbor nodes. It brings significant cost reduction compared to other scalable GNN training strategies.

Table 7: LEADING performance with GPT-2

	Cora		Arxiv	
Method	GCN	GAT	GCN	Rev-GAT
Pre-trained GPT-2	51.9 ± 1.8	54.7 ± 1.3	64.8	66.9
Supervised-FT GPT-2	70.8 ± 1.8	71.7 ± 1.9	73.2	73.8
GLEM(GPT-2) [41]	70.8 ± 1.8	71.7 ± 1.9	74.0	75.1
LEADING(GPT-2)	**80.5 ± 2.3**	**81.5 ± 1.8**	**74.7**	**75.9**

Table 8: Scalability comparison with GPT-2 on ogbn-arxiv

Methods	Memory(GB)	Running Time(S)
Supervised-FT GPT-2	26.8	15555
GIANT (GPT-2) [6]	N/A	N/A
GLEM (GPT-2) [41]	26.8	82930
LEADING (GPT-2)	27.1	27920

3.4 LEADING with LLMs

GPT-2 In this section, we extend our investigation of LEADING by incorporating LLMs, which are usually decoder-only models and are larger and different from the encoder-only LMs used in our main results. First, we perform fine-tuning on Cora and ogbn-arxiv datasets using GPT-2. Unlike the encoder-only models, which utilize the CLS (classification) embedding from the last hidden states of fine-tuned LMs as text embeddings, we utilize the information from the last token, as it encapsulates all the necessary details for prediction, aligning with the generative nature of decoder-only models. Correspondingly, we pad the sequence on the left.

The results shown in Table 7 and 8 indicate that LEADING effectively fine-tunes GPT-2 to achieve better performance, which is consistent with our experiments on other language models in Table 2. Regarding the computation cost, LEADING is capable of maintaining computational costs nearly identical to supervised fine-tuning of GPT-2 without graphs. The additional running time arises due to the sequential execution of two pipelines in LEADING, yet this can be effectively mitigated through parallel computing. It incurs significantly less computation overhead or memory cost compared to baselines such as GLEM. We similarly report N/A for GIANT due to the identical rationale analyzed in Section 3.2.

It's crucial to emphasize that enhancing model size may not be essential for achieving superior performance. The effectiveness of fine-tuning depends on multiple factors beyond just model size.

Llama-3 We repeated the same experiments on the ogbn-arxiv dataset using the newer LLMs Llama 3.2 (1B and 3B). We report the accuracy, running time, and memory cost for both LEADING and GLEM. The results in Tables 9 and 10 demonstrate that LEADING with Llama still delivers superior performance and achieves good efficiency. Notably, Llama 3.2 3B attains SOTA performance. It also maintains computational costs nearly identical to those of supervised fine-tuning of LLMs without graphs. Moreover, LEADING incurs significantly less computational time and comparable memory costs compared to GLEM, which is consistent with the conclusion in our submission and further demonstrates the advantages of LEADING.

Table 9: LEADING performance with Llama3

	3.2-1B		3.2-3B	
Method	GCN	Rev-GAT	GCN	Rev-GAT
Pre-trained Llama	65.6	67.3	67.0	68.8
Supervised-FT Llama	74.2	75.0	74.9	75.6
GLEM (Llama) [41]	75.3	76.3	75.8	76.7
LEADING (Llama)	**76.0**	**76.9**	**76.2**	**77.2**

Table 10: Scalability comparison with Llama 3 on ogbn-arxiv

	3.2-1B		3.2-3B	
Method	Memory(GB)	Time(s)	Memory(GB)	Time(s)
Supervised-FT Llama	25.7	10172	63.4	27642
GIANT (Llama) [6]	N/A	N/A	N/A	N/A
GLEM (Llama) [41]	25.7	48953	63.5	130219
LEADING (Llama)	25.7	16373	63.4	43120

3.5 Link Prediction

We conducted additional experiments on link prediction tasks, primarily following the PyTorch Geometric implementation[2]. We ran the proposed algorithm on the small Cora dataset and the large OGBL-Citation2 dataset. For Cora, we partitioned the edges into two split ratios: (1) a low ratio of 10/30/60 for train/valid/test, and (2) a high ratio of 85/5/10. We use GCN as the downstream GNN. We used GCN as the downstream GNN. For OGBL-Citation2, we followed the settings in [9] and employed a two-layer GraphSAGE model as the downstream GNN. For simplicity, we selected DistilBERT and another popular lightweight model, Sentence-BERT [31], as our language models. As shown in Tables 11 and 12, LEADING

[2]https://github.com/pyg-team/pytorch_geometric/tree/master/examples

outperforms all baselines, especially under limited training data. These results reinforce our observations on node classification and underscore the versatility of the proposed method.

Table 11: LEADING performance of link prediction on OGBL-Citation2

Method	Shallow	Pre-trained	Supervised-FT	LEADING
SBERT	77.3	81.8	83.1	**85.2**

3.6 Ablation Study

Table 12: Link prediction on Cora

Method	Low	High
Shallow Embedding	79.7	94.9
Pre-trained DistillBERT	64.7	68.7
Pre-trained SBERT	80.3	95.0
Supervised-FT DistillBERT	66.3	89.4
Supervised-FT SBERT	80.8	95.3
LEADING(DistillBERT)	81.8	95.2
LEADING(SBERT)	**83.3**	**95.7**

The necessity and importance of two mechanisms. We provide an ablation study to elucidate the individual contributions of the techniques in terms of the computation efficiency. We employ the APPNP as GNN backbones and DeBERTa as the LM for our experiments on ogbn-arxiv. We conduct tests with different layer L in four settings: (1) DeBERTa + APPNP (batch size 32); (2) DeBERTa + APPNP + Neighbor Decoupling (denoted as ND) (batch size 32); (3) DeBERTa + Implicit Modeling (denoted as IM) + Neighbor Decoupling (batch size 32); (4) DeBERTa + Implicit Modeling + Neighbor Decoupling (batch size 128).

For consistency, we sample two-hop neighbors (5 neighbors for each hop). The obtained results in Table 13 reveal that both neighbor decoupling and implicit modeling play crucial roles in reducing memory costs. Regarding neighbor decoupling, it significantly reduces memory costs without sacrificing performance. The additional running time is attributed to the sequential execution of two pipelines in our algorithm. This issue can be easily mitigated through parallel training, as described in Section 3.2. Due to the relatively small target nodes batch size of 32 used and only target nodes requiring gradients, simply adding layers will not lead to an obvious increase in memory usage for rows 5 to 8. Considering the implicit graph modeling, it achieves the same performance as 10 layers APPNP with only 2 propagation layers. Consequently, implicit modeling not only significantly reduces running time but also decreases memory costs. Furthermore, performance can be further enhanced by increasing the batch size. These findings underscore the advantages of implicit modeling and its ability to capture long-distance dependencies in graphs with very few propagation layers. Two mechanisms are both important.

Other Studies We present additional key studies in Appendices to provide a comprehensive analysis: (1) Staleness Analysis: Appendix H highlights a significant advantage of LEADING over existing cache-based models— it can easily alleviate the staleness issue. (2) Additional Baselines: Further comparisons with additional state-of-the-art models, which demonstrate LEADING's effectiveness in facilitating knowledge transfer between GNNs and language models, are presented in Appendix G. (3) PEFT:The versatility of LEADING is shown by its successful integration with existing Parameter Efficient Fine-Tuning (PEFT) approaches in Appendix E.

Table 13: Individual contribution towards the efficiency.

Methods	Acc (%)	Time/Epoch (s)	Mem.(MB)
LM+APPNP(L=2, BS:32)	70.8	5770	47243
LM+APPNP(L=3, BS:32)	N/A	N/A	OOM
LM+APPNP(L=5, BS:32)	N/A	N/A	OOM
LM+APPNP(L=10, BS:32)	N/A	N/A	OOM
LM+APPNP + ND(L=2, BS:32)	70.8	12117	8522
LM+APPNP + ND(L=3, BS:32)	71.3	18215	8522
LM+APPNP + ND(L=5, BS:32)	72.1	30311	8522
LM+APPNP + ND(L=10, BS:32)	72.8	60680	8522
LM + ND + IM(L=2, BS:32)	72.9	12117	6259
LM + ND + IM(L=2, BS:128)	73.6	2996	25103

4 Related Work

Basic structure of LMs integrated with GNNs. Several approaches have recently emerged to enhance transformer structures or graph representation techniques. Some of these methods incorporate graph structure information into attention computation [30], while others introduce orthogonal vectors for node and edge tokens to capture structural nuances [21]. While these enhancements can be effective, they often involve complex attention mechanisms, rendering the direct representation of graph structure a challenging endeavor and significantly increasing the computation complexity of model training.

Advanced structure of LMs integrated with GNNs. Recently, researchers have explored approaches that combine LMs with graph-based techniques. Notable examples include GLEM [41], which employs iterative training as mentioned earlier, and Graphformers [19, 39], which injects GNN layers into LM layers for link prediction. However, these models have their drawbacks. They either rely on a powerful model to generate high-quality soft labels, which necessitate abundant training data, or introduce significant computational overhead. Additionally, there are other approaches like GIANT [6], which uses neighbor prediction to fuse graph into LMs, and E2EG [8], which incorporates node classification into the joint training process of GIANT. However, these models also face scalability challenges [5].

5 Conclusion

Exploring the potential of pre-trained LMs for representation learning on TAGs has been of significant interest in recent years. However, it comes with significant efficiency issues in the integration of powerful LMs and GNNs. In this work, we revisit and analyze the limitations of existing approaches with a special focus on data efficiency and computation efficiency. To resolve these limitations, this work develops a novel and efficient LM-GNN end-to-end fine-tuning algorithm (LEADING) that not only effectively adapts LMs to downstream graph learning tasks with limited labeled data but also exhibits strong scalability and efficiency. Comprehensive experiments validate its superior prediction performance and efficiency in both low labeling ratio and high labeling ratio settings.

References

[1] Shaojie Bai, J Zico Kolter, and Vladlen Koltun. 2019. Deep equilibrium models. *Advances in Neural Information Processing Systems* 32 (2019).

[2] Jianfei Chen, Jun Zhu, and Le Song. 2017. Stochastic training of graph convolutional networks with variance reduction. *arXiv preprint arXiv:1710.10568* (2017).

[3] Runjin Chen, Tong Zhao, Ajay Jaiswal, Neil Shah, and Zhangyang Wang. 2024. Llaga: Large language and graph assistant. *arXiv preprint arXiv:2402.08170* (2024).

[4] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. 2018. Neural ordinary differential equations. *Advances in neural information processing systems* 31 (2018).

[5] Zhikai Chen, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang, Dawei Yin, Wenqi Fan, Hui Liu, et al. 2023. Exploring the potential of large language models (llms) in learning on graphs. *arXiv preprint arXiv:2307.03393* (2023).

[6] Eli Chien, Wei-Cheng Chang, Cho-Jui Hsieh, Hsiang-Fu Yu, Jiong Zhang, Olgica Milenkovic, and Inderjit S Dhillon. 2021. Node feature extraction by self-supervised multi-scale neighborhood prediction. *arXiv preprint arXiv:2111.00064* (2021).

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] Tu Anh Dinh, Jeroen den Boef, Joran Cornelisse, and Paul Groth. 2022. E2EG: End-to-End Node Classification Using Graph Topology and Text-based Node Attributes. *arXiv preprint arXiv:2208.04609* (2022).

[9] Keyu Duan, Qian Liu, Tat-Seng Chua, Shuicheng Yan, Wei Tsang Ooi, Qizhe Xie, and Junxian He. 2023. Simteg: A frustratingly simple approach improves textual graph learning. *arXiv preprint arXiv:2308.02565* (2023).

[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[11] Matthias Fey, Jan E Lenssen, Frank Weichert, and Jure Leskovec. 2021. Gnnautoscale: Scalable and expressive graph neural networks via historical embeddings. In *International conference on machine learning*. PMLR, 3294–3304.

[12] Fangda Gu, Heng Chang, Wenwu Zhu, Somayeh Sojoudi, and Laurent El Ghaoui. 2020. Implicit graph neural networks. *Advances in Neural Information Processing Systems* 33 (2020), 11984–11995.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[14] William L Hamilton. 2020. Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14, 3 (2020), 1–159.

[15] Zellig Harris. 1954. Distributional structure. *Word* 10, 2-3 (1954), 146–162. doi:10.1007/978-94-009-8467-7_1

[16] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. 2020. Deberta: Decoding-enhanced bert with disentangled attention. *arXiv preprint arXiv:2006.03654* (2020).

[17] Xiaoxin He, Xavier Bresson, Thomas Laurent, and Bryan Hooi. 2023. Explanations as Features: LLM-Based Features for Text-Attributed Graphs. *arXiv preprint arXiv:2305.19523* (2023).

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

[19] Bowen Jin, Wentao Zhang, Yu Zhang, Yu Meng, Xinyang Zhang, Qi Zhu, and Jiawei Han. 2023. Patton: Language model pretraining on text-rich networks. *arXiv preprint arXiv:2305.12268* (2023).

[20] Wei Jin, Haitao Mao, Zheng Li, Haoming Jiang, Chen Luo, Hongzhi Wen, Haoyu Han, Hanqing Lu, Zhengyang Wang, Ruirui Li, et al. 2023. Amazon-M2: A Multilingual Multi-locale Shopping Session Dataset for Recommendation and Text Generation. *arXiv preprint arXiv:2307.09688* (2023).

[21] Jinwoo Kim, Dat Nguyen, Seonwoo Min, Sungjun Cho, Moontae Lee, Honglak Lee, and Seunghoon Hong. 2022. Pure transformers are powerful graph learners. *Advances in Neural Information Processing Systems* 35 (2022), 14582–14595.

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

[23] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997* (2018).

[24] Guohao Li, Matthias Müller, Bernard Ghanem, and Vladlen Koltun. 2021. Training graph neural networks with 1000 layers. In *International conference on machine learning*. PMLR, 6437–6449.

[25] Yichuan Li, Kaize Ding, and Kyumin Lee. 2023. GRENADE: Graph-Centric Language Model for Self-Supervised Representation Learning on Text-Attributed Graphs. *arXiv preprint arXiv:2310.15109* (2023).

[26] Yao Ma, Xiaorui Liu, Tong Zhao, Yozen Liu, Jiliang Tang, and Neil Shah. 2021. A unified view on graph neural networks as graph signal denoising. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 1202–1211.

[27] Costas Mavromatis, Vassilis N Ioannidis, Shen Wang, Da Zheng, Soji Adeshina, Jun Ma, Han Zhao, Christos Faloutsos, and George Karypis. 2023. Train Your Own GNN Teacher: Graph-Aware Distillation on Textual Graphs. *arXiv preprint arXiv:2304.10668* (2023).

[28] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. 2000. Automating the construction of internet portals with machine learning. *Information Retrieval* 3 (2000), 127–163.

[29] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[30] Wonpyo Park, Woonggi Chang, Donggeon Lee, Juntae Kim, and Seung-won Hwang. 2022. Grpe: Relative positional encoding for graph transformer. *arXiv preprint arXiv:2201.12787* (2022).

[31] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[32] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108* (2019).

[33] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. 2008. Collective classification in network data. *AI magazine* 29, 3 (2008), 93–93.

[34] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. 2024. Graphgpt: Graph instruction tuning for large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 491–500.

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

[36] Luyu Wang, Yujia Li, Ozlem Aslan, and Oriol Vinyals. 2021. WikiGraphs: A Wikipedia text-knowledge graph paired dataset. *arXiv preprint arXiv:2107.09556* (2021).

[37] Rui Xue, Haoyu Han, MohamadAli Torkamani, Jian Pei, and Xiaorui Liu. 2023. LazyGNN: Large-Scale Graph Neural Networks via Lazy Propagation. *arXiv preprint arXiv:2302.01503* (2023).

[38] Rui Xue, Tong Zhao, Neil Shah, and Xiaorui Liu. 2024. Haste Makes Waste: A Simple Approach for Scaling Graph Neural Networks. *arXiv preprint arXiv:2410.05416* (2024).

[39] Junhan Yang, Zheng Liu, Shitao Xiao, Chaozhuo Li, Defu Lian, Sanjay Agrawal, Amit Singh, Guangzhong Sun, and Xing Xie. 2021. GraphFormers: GNN-nested transformers for representation learning on textual graph. *Advances in Neural Information Processing Systems* 34 (2021), 28798–28810.

[40] Wentao Zhang, Ziqi Yin, Zeang Sheng, Yang Li, Wen Ouyang, Xiaosen Li, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Graph attention multi-layer perceptron. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4560–4570.

[41] Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. 2022. Learning on large-scale text-attributed graphs via variational inference. *arXiv preprint arXiv:2210.14709* (2022).

[42] Yun Zhu, Yaoke Wang, Haizhou Shi, and Siliang Tang. 2024. Efficient tuning and inference for large language models on textual graphs. *arXiv preprint arXiv:2401.15569* (2024).

A Proof of Eq.7

Proof. From Eq.2, we can easily derive:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{X}_{\text{in}}} = \alpha \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*} \left(\mathbf{I} - (1-\alpha)\tilde{\mathbf{A}}\right)^{-1}, \text{where} \frac{\partial \mathcal{L}}{\partial \mathbf{X}_{\text{in}}} = \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*} \frac{dX_*}{dX_{\text{in}}} \tag{8}$$

Then, we have:

$$\left(\frac{dX_*}{dX_{\text{in}}}\right)^T \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*} = \alpha \left(\mathbf{I} - (1-\alpha)\tilde{\mathbf{A}}\right)^{-T} \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*} \tag{9}$$

For simplicity, suppose $y = \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*}$ and $G = \alpha \left(\mathbf{I} - (1-\alpha)\tilde{\mathbf{A}}\right)^{-T} \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*}$:

$$\left(\mathbf{I} - (1-\alpha)\tilde{\mathbf{A}}\right)^T G = \alpha y \tag{10}$$

Note that this is also a fixed-point equation, which we choose to solve using an iterative procedure similar to forward propagation. This approach yields Eq.7, where $G_L = \frac{\partial \mathcal{L}}{\partial \mathbf{X}_L} \approx \frac{\partial \mathcal{L}}{\partial \mathbf{X}_*}$. □

B Fixed Point Solver

Proof. We can rewrite the implicit function as

$$g(X_s, X_{\text{in}}) \;=\; f(X_s, X_{\text{in}}) \;-\; X_s \;=\; 0. \tag{11}$$

Specifically, we are first motivated by the common graph signal denoising problem:

$$\min_X \left\|X - X_{\text{in}}\right\|_F^2 \;+\; \left(\tfrac{1}{\alpha} - 1\right) \text{tr}\Big(X^\top (I - A)X\Big). \tag{12}$$

Then:

$$g(X_s, X_{\text{in}}) \;=\; X_s - X_{\text{in}} \;+\; \left(\tfrac{1}{\alpha} - 1\right)(I - A)\, X_s \;=\; 0. \tag{13}$$

We can employ any solvers such as fixed-point iteration, bisection, Newton's or quasi-Newton methods, where the root is the fixed point. Since all other solvers either require derivative calculations or converge slowly, the simplest way is to express it as the following iterative equation:

$$X_{t+1} \;=\; (1 - \alpha)\, A\, X_t \;+\; \alpha\, X_{\text{in}}, \tag{14}$$

which exactly corresponds to APPNP. This aligns with our intuition, since GNNs naturally involve such iterative aggregation. □

C End-to-end Transfer

We show how full end-to-end co-training enables more effective transfer of graph knowledge —something that simple co-training baselines cannot achieve—and how this leads to the substantial performance gains of our approach. Let the dataset and the model prediction function be

$$\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n, \quad f_{\theta,\phi}(x) = \big(g_\phi \circ h_\theta\big)(x), \tag{15}$$

where:

- $h_\theta(x)$ is the embedding of input text feature x produced by the LLM (parameters θ),
- $g_\phi(h)$ is the GNN module (parameters ϕ) that aggregates graph information based on embedding h.

Define the pointwise loss $L\big(y,\ f_{\theta,\phi}(x)\big)$. The overall training objective is

$$\min_{\theta,\phi}\ J(\theta,\phi) \;=\; \frac{1}{n}\sum_{i=1}^n L\big(y_i,\ \big(g_\phi \circ h_\theta\big)(x_i)\big). \tag{16}$$

Proof. **1. Cascaded Training without end-to-end fine-tuning**
Stage 1: LLM training (GNN fixed)

$$\min_\theta\ J_{\text{LLM}}(\theta) \;=\; \frac{1}{n}\sum_{i=1}^n L\big(y_i,\ (g_{\phi_{\text{init}}} \circ h_\theta)(x_i)\big), \tag{17}$$

where ϕ_{init} is held constant. Its gradient is

$$\nabla_\theta J_{\text{LLM}}(\theta) = \frac{1}{n}\sum_{i=1}^n \frac{\partial L\big(y_i,\ (g_{\phi_{\text{init}}} \circ h_\theta)(x_i)\big)}{\partial h_\theta(x_i)}\ \frac{\partial h_\theta(x_i)}{\partial \theta}. \tag{18}$$

Stage 2: GNN training (LLM fixed)

$$\min_\phi\ J_{\text{GNN}}(\phi) \;=\; \frac{1}{n}\sum_{i=1}^n L\big(y_i,\ (g_\phi \circ h_{\theta_{\text{fixed}}})(x_i)\big), \tag{19}$$

where θ_{fixed} is held constant. Its gradient is

$$\nabla_\phi J_{\text{GNN}}(\phi) = \frac{1}{n}\sum_{i=1}^n \frac{\partial L\big(y_i,\ (g_\phi \circ h_{\theta_{\text{fixed}}})(x_i)\big)}{\partial g_\phi(h)}\ \frac{\partial g_\phi(h)}{\partial \phi}. \tag{20}$$

In this scheme the GNN's gradient term does not back-propagate into the LLM parameters θ.

2. Advantages of LEADING's End-to-End fine-tuning
In LEADING, we optimize both parameter sets simultaneously:

$$\min_{\theta,\phi} J_{\text{LEADING}}(\theta,\phi) \;=\; \frac{1}{n}\sum_{i=1}^n L\big(y_i,\ (g_\phi \circ h_\theta)(x_i)\big). \tag{21}$$

The gradients are:
Gradient w.r.t. LLM parameters θ

$$\nabla_\theta J_{\text{LEADING}}(\theta,\phi) = \frac{1}{n}\sum_{i=1}^n \frac{\partial L\big(y_i,\ (g_\phi \circ h_\theta)(x_i)\big)}{\partial (g_\phi \circ h_\theta)(x_i)}\ \frac{\partial (g_\phi \circ h_\theta)(x_i)}{\partial h_\theta(x_i)}\ \frac{\partial h_\theta(x_i)}{\partial \theta}. \tag{22}$$

Gradient w.r.t. GNN parameters ϕ

$$\nabla_\phi J_{\text{LEADING}}(\theta,\phi) = \frac{1}{n}\sum_{i=1}^n \frac{\partial L\big(y_i,\ (g_\phi \circ h_\theta)(x_i)\big)}{\partial (g_\phi \circ h_\theta)(x_i)}\ \frac{\partial (g_\phi \circ h_\theta)(x_i)}{\partial \phi}. \tag{23}$$

The term $\frac{\partial (g_\phi \circ h_\theta)(x_i)}{\partial h_\theta(x_i)}$ captures the GNN's sensitivity to changes in the LLM embedding $h_\theta(x_i)$ and is *directly* back-propagated into θ. This aligns both modules with the final objective. □

D Variance Reduction

We theoretically show why LEADING achieves much smaller variance—even approaching zero—than sampling-based methods. We align our notation, assumptions and basic proof with VR-GCN [2] for simplicity.

Variance under standard neighbor sampling. In node-wise sampling, one obtains

$$Var_{\widehat{\mathcal{N}}^{(l)}(u)}\big[NS_u^{(l)}\big] = \frac{C_u^{(l)}}{2D^{(l)}} \sum_{v_1 \in \mathcal{N}(u)} \sum_{v_2 \in \mathcal{N}(u)} \big(P_{uv_1}\, h_{v_1}^{(l)} - P_{uv_2}\, h_{v_2}^{(l)}\big)^2. \tag{24}$$

Proof.

$$Var_{\widehat{\mathcal{N}}^{(l)}(u)}\Big[\frac{n(u)}{D^{(l)}} \sum_{v \in \widehat{\mathcal{N}}^{(l)}(u)} x_v\Big] = \Big(\frac{n(u)}{D^{(l)}}\Big)^2 \Big\{\sum_{i=1}^{D^{(l)}} Var[x_{v_i}] + \sum_{i \neq j} Cov[x_{v_i}, x_{v_j}]\Big\}$$

$$= \frac{n(u)}{D^{(l)}}\Big(1 - \frac{D^{(l)} - 1}{n(u) - 1}\Big)\Big(\sum_{v \in \mathcal{N}(u)} x_v^2 - n(u)\, \bar{x}^2\Big) = \frac{C_u^{(l)}}{2D^{(l)}} \sum_{v_1, v_2 \in \mathcal{N}(u)} (x_{v_1} - x_{v_2})^2.$$

□

Variance under LEADING. Because LEADING maintains a cached embedding $\bar{h}_v^{(l)}$, only the *difference* $\Delta h_v^{(l)} = h_v^{(l)} - \bar{h}_v^{(l)}$ is random. This term is also named as staleness in our paper. Hence

$$Var_{\widehat{\mathcal{N}}^{(l)}(u)}\big[NS_u^{(l)}\big] = \frac{C_u^{(l)}}{2D^{(l)}} \sum_{v_1, v_2 \in \mathcal{N}(u)} \big(P_{uv_1}\, \Delta h_{v_1}^{(l)} - P_{uv_2}\, \Delta h_{v_2}^{(l)}\big)^2, \tag{25}$$

since the cached term $\sum_{v \in \mathcal{N}(u)} P_{uv}\, \bar{h}_v^{(l)}$ is deterministic and contributes no variance. By updating the cache sufficiently often, we ensure $\Delta h_v^{(l)} \to 0$, which drives the variance to zero. Ordinary neighbor sampling only achieves zero variance under full-batch sampling, which is infeasible in LM+GNN co-training.

E PEFT

We illustrate the versatility of our proposed LEADING approach by integrating it with existing Parameter Efficient Fine Tuning (PEFT) approaches. Specifically, we opt for LoRA, a proven and effective method for fine-tuning the model to enhance performance. We choose to tune query, key, value, dense layer and linear layer using LoRA. From the Table 14, we observe that LoRA can further enhance performance in conjunction with our proposed LEADING algorithm.

Table 14: Prediction accuracy (%) with PEFT.

	Cora		Pubmed	
Methods	GCN	GAT	GCN	GAT
Shallow Embedding	82.0 ± 0.7	82.3 ± 0.7	78.9 ± 1.0	77.7 ± 0.9
Supervised-FT BERT	77.3 ± 1.7	78.2 ± 1.4	68.6 ± 1.8	68.6 ± 1.4
LEADING (BERT)	80.5 ± 0.4	81.6 ± 0.3	79.1 ± 0.5	79.0 ± 1.0
LEADING (LoRA)	**82.3 ± 0.5**	**82.5 ± 0.6**	**81.0 ± 0.6**	**79.5 ± 0.8**

F High Labeling Rate

We include an extra experiment to demonstrate that the proposed approach is also effective in scenarios with a high labeling rate for the Cora dataset. To verify this, we set 60% of the nodes as training set, 20% as the validation set, and the remaining 20% as the test set. The DeBERTa trained by LEADING clearly outperforms all the baselines, which verifies our conclusions as shown in Table 15.

Table 15: Accuracy (%) on Cora in high labeling rate setting.

Methods	GCN	GAT
Shallow Embedding	90.9 ± 2.7	90.6 ± 3.0
Pre-trained DeBERTa	65.9 ± 2.0	79.7 ± 3.2
Supervised-FT DeBERTa	85.9 ± 2.3	86.5 ± 1.9
GLEM (DeBERTa)	89.1 ± 0.7	89.0 ± 0.6
LEADING (DeBERTa)	**92.5 ± 2.3**	**93.2 ± 1.8**

Compared with Table 2, we point out that the performance of GLEM highly depends on the labeling ratio. In all of our experiments, GLEM works pretty well for the cases of high labeling ratios. However, The underlying reason for the above phenomenon is that GLEM adopts a two-stage approach (instead of the end-to-end training as our LEADING algorithm): (1) generate pseudo labels and (2) supervised fine-tuning of LMs on the generated pseudo labels. Therefore, the effectiveness of supervised tuning of LMs on the generated pseudo labels heavily relies on the quality of those pseudo labels. In the common low labeling data split of Cora, the quality of pseudo labels is low such that fine-tuning of GLEM using these low-quality pseudo labels will even harm the accuracy.

G Comparison with Other Paradigms

Joint Training Paradigms We further include GraphFormers [39], Patton [19] and one state-of-the-art model, Grenade [25], in our performance comparison. Specifically, GraphFormers facilitate co-training of pre-trained LMs and GNNs by injecting GNN layers into the language model's architecture. Patton also uses GraphFormers. GRENADE jointly optimizes GNN and LM encoders by employing graph-centric contrastive learning and dual-level graph-centric knowledge alignment. We follow the same experimental settings as in their original papers. Note that, since GraphFormers are originally designed for link prediction, we keep our implementation settings for node classification the same as in [41]. The results are shown in Table 16. We can observe that LEADING outperforms these joint training paradigms on both datasets. We conclude that this is because their designs are not as effective as our LEADING algorithm. They may also introduce additional problems. For example, GraphFormer must repeatedly encode all neighbors, leading to redundancy and scalability issues. These results demonstrate the superiority of our design.

Table 16: Joint training paradigms comparison(%)

Methods	Arxiv	Products
GraphFormers	72.8 ± 0.20	74.7 ± 0.16
Grenade	75.0 ± 0.19	83.1 ± 0.56
Patton	74.1 ± 0.12	77.8 ± 0.58
LEADING	**76.0 ± 0.05**	**84.1 ± 0.32**

SOTA Baseline In Table 17, we include several other strong baseline, ENGINE [42], GraphGPT [34], LLAGA [3] and SimTEG [9] which has recently demonstrated excellent performance on the ogbn-leaderboard. The results show that LEADING outperforms all baselines—none of which can achieve end-to-end training—thereby supporting our claim in the main text.

Table 17: Comparison(%) with more SOTA baselines

Dataset	ENGINE	GraphGPT	LLAGA	SimTEG	LEADING
Arxiv	76.0	75.1	75.0	77.0	**77.6**
Products	80.1	—	82.1	85.0	**86.5**

H Staleness Analysis

Staleness in cache-based models: To better isolate the staleness issue inherent in all other cache-based methods, we provide a performance comparison in Table 18. From the table, we observe that cache-based methods (eg. VR-GCN, MVS-GCN, GAS, GraphFM) can outperform classical GNNs on small datasets such as ogbn-arxiv, where the staleness problem is mitigated by the rapid refresh of the memory table. However, on large-scale datasets like ogbn-products, they perform significantly worse than scalable GNNs such as GraphSAGE, demonstrating the considerable negative impact of staleness on these models. However, our model easily outperforms all existing scalable GNNs and cache-based models, especially on large-scale datasets. This success is attributed to our two-pipeline design, where the second pipeline effectively refreshes the cache by controlling the batch size to regulate the update frequency. This demonstrates that our proposed method not only enables end-to-end training on TAGs with high scalability but also resolves the staleness issue inherent in all cache-based methods.

Table 18: Accuracy comparison.

GNNs	Reddit	Arxiv	Products
GraphSAGE	95.4	71.5	78.7
FastGCN	93.7	—	—
LADIES	92.8	—	—
Cluster-GCN	96.6	—	79.0
GraphSAINT	97.0	—	79.1
SGC	96.4	—	—
VR-GCN	94.1	71.6	76.4
MVS-GNN	94.9	71.6	76.9
GAS	96.0	71.9	77.1
GraphFM	96.2	72.1	77.2
ND+IM	96.5	72.8	80.1