Integrating Graph Transformers into PyG

CSCE 689: Graph Machine Learning – Spring 25

Omar Khater
Department of Electrical and Computer Engineering
Texas A&M University
omarkhater@tamu.edu

May 5, 2025

Abstract

Graph Transformers are emerging as powerful alternatives to classical message-passing networks, offering global attention and flexible inductive biases for learning on structured data. However, existing implementations in PyTorch Geometric (PyG) remain fragmented—lacking unified support for structural biases, positional encodings, and upstream-ready integration.

In this project, we develop a modular and extensible `GraphTransformer` layer for PyG that supports full token-wise attention, configurable bias providers, native positional encodings, and TorchScript compatibility. Our implementation spans two repositories: `graph-transformer-core`, which encapsulates the reusable model layer and supporting utilities with test-driven design to be submitted as pull request for PyG; and `graph-transformer-benchmark`, which provides a training pipeline, experiment tracking (via MLflow), and standardized evaluation using Hydra.

We benchmark our layer against standard GNNs (GCN, GAT, GIN, GraphSAGE) on representative datasets (MUTAG, CORA, PubMed), achieving strong baseline accuracy even without specialized enhancements. Future work includes Bayesian hyperparameter tuning, multi-graph batching, and upstream contribution to `torch_geometric.contrib.nn.models`. Our work bridges architectural completeness with engineering rigor, laying the foundation for a unified Graph Transformer module in the PyG ecosystem.

Keywords

Graph Neural Networks, Transformers, Structural Bias, Positional Encoding, PyTorch Geometric, Software Engineering, Benchmarking

1 Introduction

Graph Neural Networks (GNNs) have become the de facto standard for learning on relational data, leveraging message-passing mechanisms to aggregate neighborhood information and model structural dependencies. In parallel, Transformer architectures have transformed the landscape of deep learning through self-attention mechanisms capable of capturing global context, first in natural language processing and now across domains like vision and audio.

Recent advances suggest that combining the localized inductive biases of GNNs with the expressive power of Transformers can yield models that generalize better across both node-level and graph-level tasks. In particular, augmenting attention mechanisms with structural information—such as spatial distances or node degrees—has proven effective in bridging the gap between global modeling and graph topology.

Our work is inspired by the design taxonomy proposed by Bai and Wang [1], which highlights three key dimensions for Graph Transformer design: the use of auxiliary GNN layers, positional encoding strategies, and attention biasing mechanisms. This framework motivates our design of a modular Graph Transformer layer that is both expressive and extensible.

Despite growing academic interest, most implementations of Graph Transformers remain isolated in standalone repositories, often lacking compatibility with standard libraries like PyTorch Geometric (PyG). Practitioners are frequently forced to circumvent PyG's native abstractions, creating friction in experimentation and integration. To address this gap, we contribute a PyG-native `GraphTransformer` module with the following features:

- A fully modular Graph Transformer model that supports token-wise self-attention, learnable structural biases (e.g., edge features, hop count, spatial distance), and an optional super-node for global readout.
- A plug-and-play API for injecting user-defined bias providers and positional encoders—without modifying PyG's data pipelines. Optional GNN blocks can be inserted before, after, or in parallel with the attention layers.
- A separate benchmarking suite (`benchmark_graph_transformer`) that enables end-to-end training, MLflow-based logging, and reproducible evaluation on standard benchmarks.

Together, these contributions aim to unify disparate efforts in Graph Transformer research into a clean, reusable, and upstream-ready codebase for the PyG ecosystem.

The remainder of this report is organized as follows: Section 2 surveys related work in Graph Transformers and positional encoding strategies. Section 3 outlines our architecture and implementation details. Section 4 presents initial benchmarking results and software quality metrics. Finally, Section 5 offers closing thoughts and outlines future directions for the project.

2 Related Work

Graph Neural Networks (GNNs) are the predominant approach for learning from structured graph data. GNNs rely on local message passing, iteratively updating node embeddings based on aggregated neighbor information. While highly effective for encoding local structures, this mechanism inherently limits their receptive field, making it challenging to capture long-range dependencies across distant nodes, often causing the well-known over-smoothing phenomenon [2].

In contrast, **Transformers** were initially introduced for sequence modeling in Natural Language Processing (NLP). Transformers employ a self-attention mechanism, enabling global interactions among all tokens in the input sequence, thus directly modeling long-range dependencies. This flexibility, driven by query-key-value projections and residual connections, has facilitated the adaptation of Transformers across diverse modalities such as vision, audio, and graph-structured data [1]. Figure 2 illustrates the core Transformer components relevant to graph-based modeling.

Trade-off: GNN vs. Transformer. Drawing parallels from the computer vision domain, GNNs resemble convolutional neural networks (CNNs), characterized by localized operations, efficient computation, and strong inductive biases. Transformers, conversely, behave akin to fully connected layers, offering increased representational power at the cost of higher computational complexity.

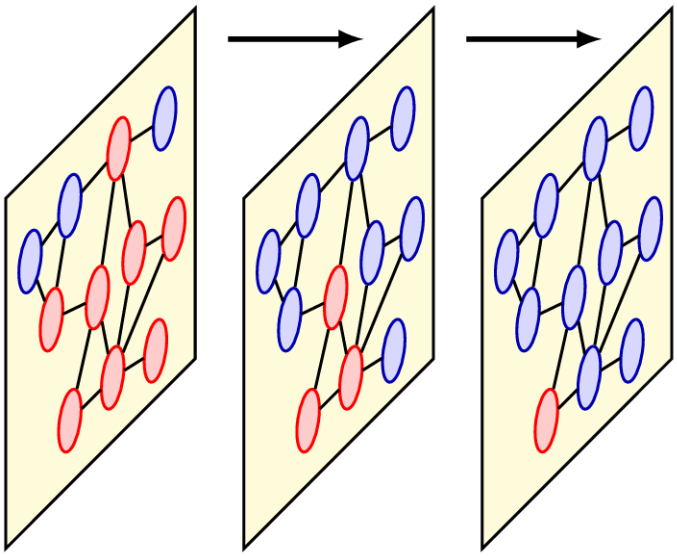
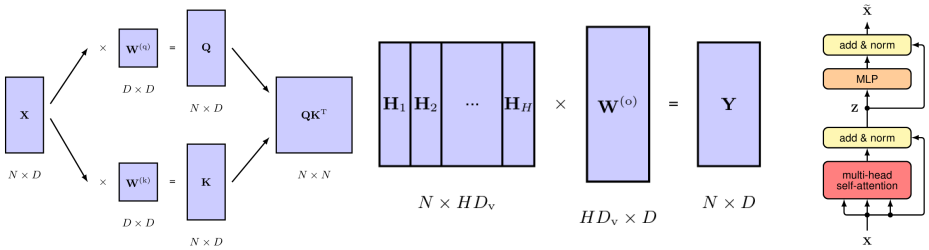


Figure 1: Expansion of receptive fields in message-passing GNNs. Information from distant nodes is aggregated through multiple local message-passing layers. Adapted from [3].



(a) Scaled dot-product attention. (b) Multi-head aggregation. (c) Full Transformer block.

Figure 2: Transformer architecture components: (a) attention weight computation, (b) multi-head concatenation, and (c) the complete Transformer block with residual connections and normalization layers. Adapted from [3].

Graph Transformers aim to harmonize these complementary advantages, integrating global attention with structurally-informed inductive biases such as positional encodings and attention biases derived from graph topology [1, 4].

Architectural Trends. Bai and Wang [1] provide a comprehensive taxonomy of existing Graph Transformer variants. They categorize these models based on three critical design dimensions:

1. **GNN Auxiliary (GA):** Combining local GNN layers with global Transformer attention.
2. **Positional Encodings (PE):** Embedding structural or spectral node information to guide attention.
3. **Attention Bias (AT):** Modifying attention logits based on graph connectivity or structural distances.

Figure 4 summarizes prominent Graph Transformer architectures across these dimensions, highlighting their diverse approaches.

Task Types and Benchmarks. Graph learning tasks primarily fall into two categories: node-level and graph-level predictions. Node-level tasks focus on classifying individual nodes based on structural and feature information, with prominent benchmarks including citation networks (Cora, PubMed) and Open Graph Benchmark datasets such as ogbn-arxiv. Graph-level tasks involve predicting properties

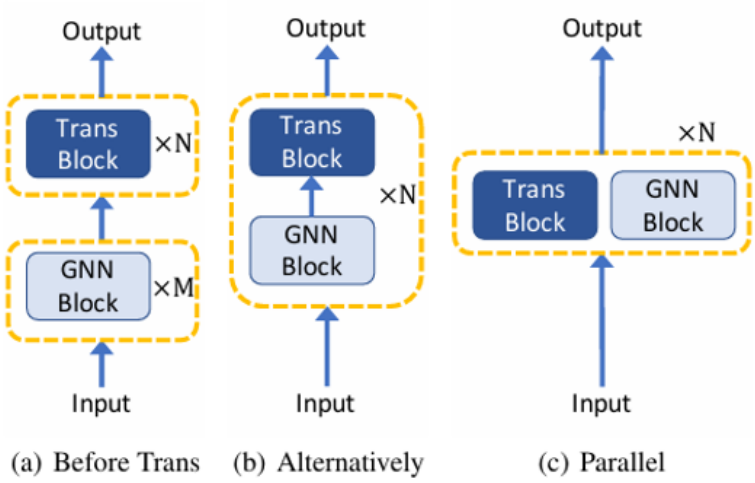


Figure 3: Integration strategies for combining GNN and Transformer modules: (a) stacked, (b) interleaved, and (c) parallel. These designs differ in how they balance local message passing and global attention. Adapted from Bai and Wang [1].

of entire graphs, common in molecular and biological classification tasks. Standard benchmarks include the TU dataset family (e.g., MUTAG, PROTEINS) and Open Graph Benchmark (OGB) molecule datasets (e.g., ogbg-molhiv, ogbg-molpcba). Figure 5 visually illustrates representative examples of these task categories.

PyTorch Geometric (PyG) [6] is a widely adopted framework offering optimized data abstractions and diverse implementations for graph-based machine learning. Although PyG includes Transformer-inspired layers such as `TransformerConv`, `HGTConv`, and `GPSConv`, there remains a clear gap in providing a unified, extensible Graph Transformer layer with integrated support for diverse structural biases and positional encodings. Addressing this gap forms the core motivation behind this project.

3 Proposed Method

3.1 Motivation and Objectives

Existing Graph Transformer layers in PyTorch Geometric (PyG) [6] only partially cover the needs of flexible graph-based modeling. Layers like `TransformerConv`, `GPSConv`, `PointTransformerConv`, and `HGTConv` differ in their support for global attention, structural priors, and deployment-readiness. To address these inconsistencies, we aim to build a unified, modular Graph-Transformer layer with broad coverage of structural biases, TorchScript support, and plug-and-play positional encodings.

Table 1: Comparison of Transformer-inspired layers in PyG.

Layer/Model	Global Attention	Structural Bias	Super-node Token	Degree/Centrality	TorchScript Ready
`TransformerConv`	**No** – only along edges	✓ Edge-feature keys/values	×	×	Partial (needs `_alpha` fix)
`GPSConv`	**Hybrid** – MPNN + global attention	PE added before attention	×	External preproc	Partial (Python-heavy modules)
`PointTransformerConv`	**No** – k-NN / radius	Relative 3D kernel	×	×	✓ (doc-supported)
`HGTConv`	**No** – per relation	Relation-specific keys/queries	×	×	Partial (complex typing)

3.2 Design Goals and Implementation

We introduce a new `GraphTransformer` layer that addresses the fragmentation in current implementations. Our layer satisfies the following:

- **Full global attention** across all node pairs.

Method	GA	PE	AT	Code
[Zhu *et al.*, 2019]			✓	✓
[Shiv and Quirk, 2019]		✓		
[Wang *et al.*, 2019]		✓	✓	
U2GNN [Nguyen *et al.*, 2019]			✓	✓
HeGT [Yao *et al.*, 2020]			✓	✓
Graformer [Schmitt *et al.*, 2020]			✓	✓
PLAN [Khoo *et al.*, 2020]			✓	✓
UniMP [Shi *et al.*, 2020]			✓	
GTOS [Cai and Lam, 2020]		✓	✓	✓
Graph Trans [Dwivedi and Bresson, 2020]		✓	✓	✓
Grover [Rong *et al.*, 2020]	✓			✓
Graph-BERT [Zhang *et al.*, 2020]	✓	✓		✓
SE(3)-Transformer [Fuchs *et al.*, 2020]			✓	✓
Mesh Graphormer [Lin *et al.*, 2021]	✓	✓		✓
Gophormer [Zhao *et al.*, 2021]			✓	
EGT [Hussain *et al.*, 2021]		✓	✓	✓
SAN [Kreuzer *et al.*, 2021]		✓	✓	✓
GraphiT [Mialon *et al.*, 2021]	✓	✓	✓	✓
Graphormer [Ying *et al.*, 2021]	✓	✓	✓	✓
Mask-transformer [Min *et al.*, 2022]			✓	✓
TorchMD-NET [Thölke and de Fabritiis, 2022]			✓	✓

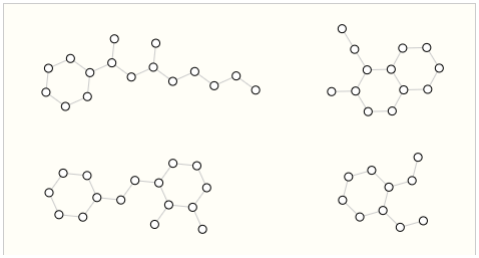
Figure 4: Landscape of Graph Transformer architectures across dimensions of GNN Auxiliary (GA), Positional Encoding (PE), and Attention Bias (AT). Adapted from Bai and Wang [1].

- **Native support for structural biases** (e.g., shortest path, edge distance, node degree).
- **Integrated positional encodings** using configurable schemes (e.g., Laplacian eigenvectors, SVD, spatial kernels).
- **Super-node token** support for graph-level representation learning.
- **TorchScript compatibility** with static and deployable graph execution.

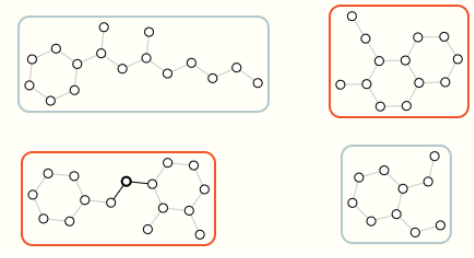
3.3 Software Engineering Approach

Our codebase emphasizes maintainability and production readiness:

- **Test-driven development (TDD)**: every module is covered by unit and smoke tests.
- **Strict modularity**: responsibilities are separated into `data/`, `models/`, `training/`, `evaluation/`.
- **Hydra-based configs**: all experiments are reproducibly configured and logged.
- **MLflow integration**: automatic logging of metrics, configs, and checkpoints.
- **Pre-commit and CI**: enforced style, typing, and correctness via `black`, `flake8`, `mypy`, and tests.



(a) Graph-level task (Input).



(b) Graph-level task (Output).

(c) Node-level task (Input).

(d) Node-level task (Output).

Figure 5: Illustrative examples of graph-level and node-level prediction tasks. Graph-level tasks involve labeling entire graphs (top row), while node-level tasks classify individual nodes (bottom row). Adapted from [5].

3.4 Initial Results and Future Plans

We validate our model's correctness and TorchScriptability on synthetic tasks and unit tests. Benchmarks on real datasets (e.g., Cora, PubMed, MUTAG) are currently underway using Bayesian optimization for hyperparameter tuning.

- **Current evaluation**: runtime correctness, forward pass validation, structural bias injection.
- **Planned work**: full dataset benchmarking with Optuna sweeps, comparison against PyG baselines.
- **Upstream intent**: the code is designed for submission via Pull Request (PR) to `torch_geometric.contrib.nn.models`.

Our method balances research extensibility with production hygiene and is positioned to unify future Graph Transformer development within the PyG ecosystem.

4 Results

We evaluate our proposed `GraphTransformer` layer across two complementary axes: (1) **Software Engineering Quality**, reflecting modular design and PyG integration standards; and (2) **Initial Benchmarking**, validating baseline functionality on real-world datasets under controlled hyperparameters.

4.1 System Configuration

All experiments were conducted on a local development machine with the following specifications:

While this environment is sufficient for prototyping and reproducibility, GPU acceleration is expected to yield faster convergence and will be included in future runs.

Table 2: Hardware and software specifications for benchmark experiments.

Component	Specification
OS	Windows 11 Education (Build 26100)
Processor	Intel Core i5-12400 (6 Cores, 12 Threads, 2.5GHz base)
RAM	16GB DDR4 (System default)
System Model	Dell Inspiron 3910
BIOS Version	Dell Inc. 1.9.1 (Jan 2023)
Platform Type	Desktop, UEFI boot
GPU	CPU-only execution (no CUDA)

4.2 Software Engineering Quality

Our implementation strictly follows PyTorch Geometric (PyG) development practices. Key highlights include:

- **Encapsulation:** Each module (e.g., bias providers, positional encoders, transformer layers) adheres to the single-responsibility principle and resides under the `graph_transformer/` directory.
- **Test Coverage:** Functional smoke tests and unit tests are integrated with GitHub Actions CI. TorchScript tracing and `num_layers=0` edge cases are explicitly covered.
- **Lint and Type Checks:** The codebase conforms to `black`, `flake8`, `isort`, and `mypy` standards, with no runtime warnings on TorchScript export.
- **Pull Request Readiness:** Naming conventions, docstrings, and factory patterns align with PyG, easing eventual upstream contribution.

Code quality metrics, summarized in Table 3, were computed to guide future refactoring. They reflect a high degree of modularity, with localized complexity in utility logic.

Table 3: Initial code quality by module. CC: Cyclomatic Complexity; MI: Maintainability Index.

Module	Max CC	Grade	Avg MI
Bias	5	*A*	72.9
Layers	9	*B*	72.5
Models	7	*B*	49.5
Positional	3	*A*	91.2
Utils	11	*C*	72.4

4.3 Initial Benchmarking

To validate correctness and ensure functional parity with common GNN models, we benchmarked classification tasks using fixed hyperparameters across all models and datasets:

Table 4: Common hyperparameter configuration.

Parameter	Value
Learning rate	1e-3
Epochs	15
Hidden dimension	64
Attention heads	4
Transformer layers	4
Dropout rate	0.1

Despite lacking any GNN or positional enhancements, the `GraphTransformer` achieves superior accuracy on CORA, PubMed and strong results on MUTAG—supporting the architectural flexibility and correctness of our implementation.

Table 5: Benchmark results on real datasets. Each model is trained for 15 epochs with fixed hyperparameters. Runtime is measured in minutes.

Task	Dataset	Model	Test Accuracy	Time (min)
Graph-level	MUTAG	GraphTransformer	**0.722**	5.8
		GIN	0.611	5.2
		GCN	0.611	5.2
		GAT	0.611	5.1
		SAGE	0.611	5.0
Node-level	CORA	GraphTransformer	**0.985**	13.3
		GIN	0.800	6.5
		GCN	0.376	5.5
		GAT	0.441	5.1
		SAGE	0.606	5.5
Node-level	PubMed	GraphTransformer	**0.793**	21.6
		GIN	0.643	6.6
		GCN	0.449	5.9
		GAT	0.482	6.2
		SAGE	0.638	6.3

5 Conclusion

In this work, we introduced a unified and extensible `GraphTransformer` layer for PyTorch Geometric (PyG), addressing key limitations in existing Transformer-style models for graphs. Our implementation supports global attention, native positional encodings, structural bias injection, and TorchScript compatibility—all encapsulated within a modular, testable, and PyG-aligned architecture.

This effort emphasizes not only functional innovation but also software engineering rigor: configuration is managed declaratively via Hydra, all training and evaluation pipelines are reproducible through MLflow tracking, and our codebase adheres to PyG's contribution and documentation standards. As a result, the model is well-suited for community reuse, extension, and potential upstream submission.

Initial benchmarks validate correctness and reveal strong early performance across representative datasets (e.g., CORA, MUTAG), even in the absence of specialized GNN or encoding components. These results establish a baseline for future architectural and empirical improvements.

Future Work

Building upon this foundation, we outline several concrete directions for continued development:

- **Bayesian Optimization:** We are launching Optuna-based hyperparameter sweeps across datasets such as CORA and `ogbn-arxiv`, exploring attention width, dropout, bias combinations, and positional encoding types.
- **API Extensions:** Planned enhancements include support for multi-graph batching, additional bias providers (e.g., motif- or flow-based), and improved type coverage for TorchScript.
- **GPU Support**: Currently, all our experiments were made on CPU. Comprehensive evaluation on GPU would be essential before pushing the pull request.
- **Upstream Contribution:** We intend to package the module for community release as a pull request to `torch_geometric.contrib.nn.models`, alongside examples and tests.

Overall, our contribution bridges the gap between Transformer generality and graph-specific architectural bias, offering a clean, extensible foundation for future graph representation learning research.

References

[1] Yunsheng Bai and Lili Wang. Transformer for graphs: An overview from architecture perspective. *arXiv preprint arXiv:2202.08455*, 2022.

[2] Vijay Prakash Dwivedi, Johannes Klicpera, et al. Graph neural networks with learnable structural and positional representations. *arXiv preprint arXiv:2110.07875*, 2021.

[3] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Springer, 2024.

[4] Zhitao Ying, Tianle Cai, Shengjie Luo, Shuxin Zheng, Guolin Ke, Di He, Yanming Shen, and Tie-Yan Liu. Do transformers really perform bad for graph representation? In *NeurIPS*, 2021.

[5] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[6] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with pytorch geometric. *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.