PyG 2.0: Scalable Learning on Real World Graphs

Matthias Fey[1]   Jinu Sunil[1]   Akihiro Nitta[1]   Rishi Puri[2]   Manan Shah[1]

Blaž Stojanovič[1]   Ramona Bendias[1]   Alexandra Barghi[2]   Vid Kocijan[1]

Zecheng Zhang[1]   Xinwei He[1]   Jan Eric Lenssen[1]   Jure Leskovec[1]

[1]Kumo.ai, [2] Nvidia

Abstract

PyG (PyTorch Geometric) has evolved significantly since its initial release, establishing itself as a leading framework for Graph Neural Networks. In this paper, we present PyG 2.0, a comprehensive update that introduces substantial improvements in scalability and real-world application capabilities. We detail the framework's enhanced architecture, including support for heterogeneous and temporal graphs, scalable feature/graph stores, and various optimizations, enabling researchers and practitioners to tackle large-scale graph learning problems efficiently. Over the recent years, PyG has been supporting graph learning in a large variety of application areas, which we will summarize, while providing a deep dive into the important areas of relational deep learning and large language modeling.

CCS Concepts

• **Computing methodologies** → **Machine learning**; • **Information systems**;

Keywords

temporal graph learning, data mining

1 Introduction

Graph Neural Networks (GNNs) have emerged as powerful tools for learning on ubiquitous graph-structured data. From social networks, knowledge bases, relational databases, to spatial graphs describing molecular structures, 3D scenes or objects, graphs are used to store most of the world's data. Since 2019, *PyG (PyTorch Geometric)* [28] has been an important cornerstone in advancing deep learning on all these different types of graphs (*cf.* Sec. 3 for a summary). PyG introduced a general message passing scheme that allows for a flexible formulation of Graph Neural Networks. This is achieved by decomposing neural message passing [34] into MESSAGE, AGGRE-GATION, and UPDATE functions that can be customized to create various types of graph-based operators, thus supporting a broad range of models in a unified framework, which can automatically be mapped onto GPUs.

In the early years, most applied research around Graph Neural Networks revolved around finding the best operators to solve small-scale benchmark tasks, such as node classification on the Cora citation graph [49, 77, 86], the graph-based equivalent to MNIST [22]. Since then, the field of graph learning has rapidly evolved, strongly supported and driven by advancements in infrastructure provided by PyG. GNNs can now be trained efficiently on web-scale, heterogeneous, temporal and multi-modal graphs, are explainable, and easily deployable for a wide range of practical applications. PyG has evolved into a comprehensive blueprint for end-to-end graph-based machine learning, enabling these functionalities.

In this work, we present the design principles and architectural decisions behind PyG, beginning with the foundational changes introduced in PyG 2.0 and extending through its continuous evolution to the current state of the library. PyG 2.0 marked a significant milestone in the library's development over three years ago, this paper encompasses the full trajectory of improvements and innovations that have been integrated into PyG up to its most recent version. We focus on the following three core aspects that have been refined and expanded throughout this evolution:

- **Heterogeneity.** Real world graphs have diverse node and edge types. PyG natively supports heterogeneous graph data types and message passing, as well as functionality for learning on temporal graphs.
- **Scaling and Efficiency.** Many use cases of graph learning have massive graphs (∼ 10 billion nodes), which need to be supported through optimized loading and training APIs. To this end, we present novel distributed processing capabilities, efficient data formats, loaders, and samplers, accelerated message passing, and compilation mechanisms.
- **Explainability.** Understanding how a model arrives at its decision is crucial in several domains and often required for trust in deep learning models deployed in practice. We discuss explainability in the heterogeneous graph learning setting and describe our plug-and-play method to make any GNN within PyG explainable out-of-the-box.

Graph learning powered by PyG has made an impact in a wide range of practical fields. To showcase its generality, we also provide an overview of applications in chemistry, material design, computer vision, weather, and traffic forecasting. Moreover, we deep-dive into two specific application areas: GNN (and PyG) integration in Large Language Models [40] and Relational Deep Learning [27].

2 🔷 PyG 2.0: End-to-End Graph Learning

In this section, we describe the building blocks that assemble the blueprint for end-to-end graph learning with PyG. We begin by providing an overview of all discussions in Sec. 2.1. Then, the subsequent sections outline the individual components in more detail, such as implementation aspects of our heterogeneous neural framework in Sec. 2.2, scaling to real world graphs in Sec. 2.3, and post-processing capabilities, *e.g.*, via explainability, in Sec. 2.4.

2.1 Framework Overview

PyG is a library built upon ⭕ PyTorch [68] to easily write and train Graph Neural Networks for a wide range of applications related to structured data. It utilizes a tensor-centric API, *i.e.* it exclusively operates on tensor-like data to define feature representations, graph structures and neural building blocks, and thus offers an intuitive experience which facilitates straightforward integration within the broader PyTorch ecosystem. This design principle allows PyG to keep up-to-date with advances of its core, *e.g.*, nested tensors[1] for handling heterogeneous data of varying size, TorchScript[2] for model serialization and deployment, torch.fx [71] for model transformations, or torch.compile[3] for model optimization via Just-In-Time (JIT) compilation. PyG leverages vectorized operations throughout its pipeline to maximize efficiency. In cases where vectorization is not feasible—such as during graph sampling—we provide specialized C++ and CUDA kernels via our external (but optional) low-level **pyg-lib** package.

Figure 1 illustrates the comprehensive architecture of PyG, highlighting its modular and plug-and-play design. The system is broadly organized into three main components: **(1)** graph infrastructure, **(2)** a neural framework, and **(3)** post-processing routines. The graph infrastructure (*cf.* Sec. 2.3) manages the lifecycle of (heterogeneous and temporal) graph data, supporting multi-modal feature processing, graph conversions, multi-threaded graph samplers, and distributed training. The neural framework (*cf.* Sec. 2.2) builds upon this data pipeline to define core interfaces and implementations for graph learning. It offers efficient support for (heterogeneous) message passing, sparse aggregation operations, GPU acceleration, and model compilation. Finally, post-processing (*cf.* Sec. 2.4) routines operate on the output of graph-based models to generate explanations, compute evaluation metrics, or perform k-nearest neighbor searches.

PyG's design offers flexibility via standardized interfaces throughout the full end-to-end pipeline. One can easily swap components independently—transitioning from in-memory storage to databases, changing sampling strategies, or updating model architectures—all without disrupting other parts of the system. The same interfaces work consistently whether we are handling small graphs or massive networks. This architecture makes PyG particularly research-friendly, as it enables easy experimentation with novel techniques at any stage of the pipeline.

---

[1]torch.nested: https://pytorch.org/tutorials/prototype/nestedtensor
[2]TorchScript: https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial
[3]torch.compile: https://pytorch.org/tutorials/intermediate/torch_compile_tutorial

2.2 Neural Framework

Message Passing Graph Neural Networks (MP-GNNs) [28, 34] are a generic framework to define a wide range of graph-based deep learning architectures. Given a graph $G = (\mathcal{V}, \mathcal{E})$ with input node embeddings $\{\mathbf{h}_v^{(0)}\}_{v \in \mathcal{V}}$ and edge embeddings $\{\mathbf{e}_{(v,w)}\}_{(v,w) \in \mathcal{E}}$, a single neural message passing step updates the node features by

$$\mathbf{h}_v^{(\ell+1)} = f\left(\mathbf{h}_v^{(\ell)}, \left\{\!\!\left\{ g\left(\mathbf{h}_w^{(\ell)}, \mathbf{e}_{(w,v)}, \mathbf{h}_v^{(\ell)}\right) \mid w \in \mathcal{N}(v) \right\}\!\!\right\}\right), \quad (1)$$

where f and g are differentiable, optimizable functions and $\{\!\!\{\cdot\}\!\!\}$ a permutation invariant set aggregator, such as mean, max, sum. PyG automatically maps all implementations of the framework efficiently to GPUs by alternating between parallelization over edges (function g) and nodes (function f). Almost all recently proposed GNN operators can be mapped to this interface, including (but not limited to) the methods already integrated into PyG [21, 34, 37, 49, 86, and many others].

Accelerated Message Passing. As the primary operation in GNNs, message passing becomes a performance bottleneck, making its efficiency essential. Within the first iteration of PyG, message passing was implemented by explicitly materializing $(\mathbf{h}_w^{(\ell)}, \mathbf{e}_{(w,v)}, \mathbf{h}_v^{(\ell)})$ into edge-level space, followed by an aggregation into node-level space using atomic operations [28]. While easy to implement and effective to parallelize, memory requirements can become a bottleneck on denser graphs.

With PyG 2.0, we introduce a new and unified way to accelerate message passing, leading to less memory-bottlenecked GNN workflows while preserving full backward compatibility. In order to achieve this, we introduce the EdgeIndex tensor, which holds pair-wise source and destination node indices in sparse Coordinate Format (COO) of shape $\{1, \ldots, |\mathcal{V}|\}^{2 \times |\mathcal{E}|}$. EdgeIndex sub-classes a general torch.Tensor, and thus preserves the ease-of-use of regular COO-based PyG workflows. However, it can hold additional (meta)data, *e.g.*, its sort order (if present) or whether edges are undirected. Furthermore, it introduces a caching mechanism for fast conversion to Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) sparse formats. Caches are filled based on demand, and are maintained and adjusted over its lifespan. As a result, message passing in PyG can now rely on this (meta)data information to choose the optimal message passing computation path: First, if the EdgeIndex is sorted by row or column, we can efficiently leverage sparse matrix multiplications (SpMMs) [94] and segmented aggregations in GNN layers. This ordering enhances data locality, reduces memory requirements, and enables greater parallelism on GPUs. Second, for repeated GNN layer execution, caching the graph's CSC and CSR formats significantly reduces overhead during the backward pass. Without this cache, computing the transposed adjacency matrix $\mathbf{A}^\top$—derived from the edge set $\mathcal{E}$—would be repeatedly required. Finally, for undirected graphs where $\mathbf{A} = \mathbf{A}^\top$, caching the CSR format becomes unnecessary, further saving memory and computation.

Aggregations as a First-Class Principle. One of the most critical components of GNNs is the choice of the aggregation function. It may account for symmetry, invariance [7, 10], and the expressive power [19, 93] of GNNs to capture different types of properties of graphs. Other works [37, 55, 99] empirically show that the choice
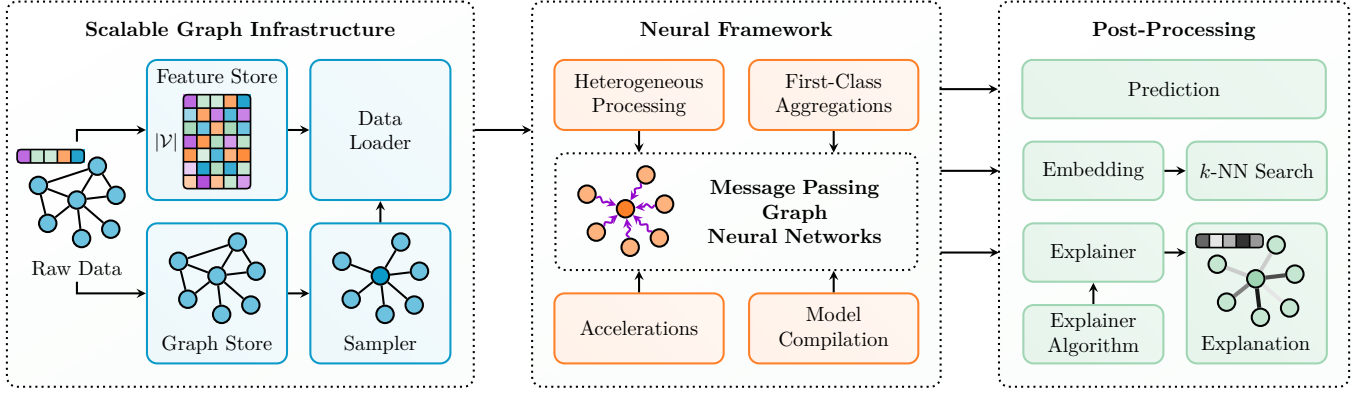
Figure 1: Architectural overview of PyG 2.0: The system's modular design allows to swap out any component without affecting other parts of the pipeline. For example, one can seamlessly change the `FeatureStore` from in-memory to distributed key-value storage without modifying the `DataLoader` or model parts. This plug-and-play approach extends throughout the framework—from storage implementations to sampling strategies and explainers. The core neural framework incorporates multiple performance optimizations including GPU accelerations, heterogeneous processing and model compilation techniques, ensuring efficient GNN training even on large-scale, heterogeneous and temporal graphs.

of aggregation function is crucial to the performance of GNNs, and even utilize multiple aggregations [19, 83] or learnable aggregations [55] to obtain substantial improvements.

Inspired by this work, we made the concept of aggregation a first-class principle in PyG 2.0, which allows users to easily plug-and-play with all kinds of aggregations—from simple ones (*e.g.*, mean, max, sum) to advanced ones (*e.g.*, median, variance, standard deviation), learnable ones [55], and unconventional ones (*e.g.*, via LSTMs [37] or equilibrium [6])—which can be also seamlessly stacked together [19, 83]. Unifying the concept of aggregation helps us to perform optimization and specialized implementations in a single place, which can be utilized within both message passing and global readouts.

Heterogeneous Message Passing. PyG 2.0 introduces enhanced support for heterogeneous graphs, allowing seamless handling of multiple node and edge types. This capability is essential for real-world applications where graphs naturally contain different types of entities and relationships.

Formally, a *heterogeneous graph* is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \phi, \psi)$, where each node $v \in \mathcal{V}$ and edge $e \in \mathcal{E}$ corresponds to a type $\phi(v) : \mathcal{V} \to \mathcal{T}$ and $\psi(e) : \mathcal{E} \to \mathcal{R}$, respectively. Then, *heterogeneous message passing* [42, 74] is a *nested* version of Eq. (1), adding an aggregation over all incoming edge types to learn distinct messages for each node type. Heterogeneity is natively supported by PyG 2.0. It provides heterogeneous data types, transformations, graph samplers, and can automatically turn any message passing GNN into a heterogeneous variant. This is achieved via a custom `torch.fx` [71] transformation, which takes in a homogeneous GNN, replicates its GNN layers for every edge type in $\mathcal{R}$, and then transforms its computation graph to perform bipartite message passing over every edge type, followed by a custom aggregation to bundle messages pointing to the same destination node type.

The main challenge to efficiently implement heterogeneous GNNs lies in the varying number of nodes that belong to each node type

$T \in \mathcal{T}$, *i.e.* node features can be understood as a set $\{\mathbf{H}_T^{(\ell)}\}_{T \in \mathcal{T}}$, $\mathbf{H}_T^{(\ell)} \in \mathbb{R}^{N_T \times F}$, where the number of nodes N_T may vary for each node type. In cases of dedicated heterogeneous GNN instantiations [42, 62, 89], PyG leverages *grouped* and *segmented* matrix multiplications to implement parallel projections across node/edge types efficiently. Such re-occurring operation in heterogeneous message passing is defined as $\{\mathbf{H}_T^{(\ell)} \mathbf{W}_T^{(\ell)}\}_{T \in \mathcal{T}}$ based on the three-dimensional weight tensor $\mathbf{W}^{(\ell)} \in \mathbb{R}^{|T| \times F \times F'}$, and requires both backward implementations w.r.t $\mathbf{H}_T^{(\ell)}$ and $\mathbf{W}^{(\ell)}$. Internally, we implement both forward and backward passes using high-performance libraries such as CUTLASS [85].

Model Compilation. PyTorch's eager mode excels during the development and debugging phase of model design. However, in production, performance—both in terms of speed and memory efficiency—becomes a top priority. PyG 2.0 supports kernel fusion via `torch.compile`, enabling end-to-end compilation without graph breaks. This allows multiple operations—including sparse computations and feature transformations—to be fused into a single, highly optimized kernel. As a result, memory access and kernel launch overheads are minimized, making message passing significantly faster, especially for deeper or wider GNNs. In order to support `torch.compile` within the irregular input workflows of PyG to full efficiency, we have revisited our entire code base to **(1)** avoid graph breaks and **(2)** remove any device synchronizations. Our `MessagePassing` interface supports `torch.compile` out-of-the-box, without any user adjustments required. On average, we observe 2-3x speedup in runtime while maintaining predictive accuracy, *cf.* Table 1.

Graph Transformers. Aligned with recent advances in graph machine learning, PyG 2.0 integrates state-of-the-art *Graph Transformer* architectures [17, 48, 70, 79, 92] into its package, applicable for learning on both many small graphs and single large graphs. Positional encodings, which capture graph topology, can be computed

Optimization	GIN	GraphSAGE	EdgeCNN	GCN	GAT
Eager (baseline)	9.56	9.45	98.51	19.73	29.72
compile	2.86	2.79	58.12	**4.62**	8.32
compile + Trim	**1.98**	**1.96**	**23.11**	5.86	**7.93**

Table 1: Forward and backward pass runtime (milliseconds) across different GNN architectures. The baseline uses default eager mode without compilation. Compilation provides 2–3× speedup. Compilation+trimming achieves 4–5× speedup. Benchmarks generated using *this script*

either during pre-processing or dynamically at runtime. These models are built on unified interfaces and seamlessly incorporate components from traditional GNN workflows.

2.3 Scalable Graph Infrastructure

Real-world graphs come in various shapes and sizes, and there is a growing interest in scaling GNNs to graphs with billions of nodes and multi-thousand-dimensional features. Such large-scale data is typically stored in external systems, *e.g.*, embedded databases—with growing interest in using frameworks like PyG to support mini-batch GNN training directly on top of these storage platforms.

To meet this need, PyG 2.0 introduces new `FeatureStore` and `GraphStore` remote backend interfaces that enable seamless interoperability with custom storage, all while maintaining the familiarity of the PyG training loop and core PyTorch abstractions.

For large or distributed graphs, in which node features and edge indices are stored in custom locations, users are only required to implement the relevant methods within the remote backend interface; the rest of the training loop looks identical to an in-memory implementation, and any distributed communication required is handled transparently by PyG.

Support for custom feature and graph storages in PyG 2.0 is enabled by defining a clear separation of concerns within the library. Concretely, the data loading loop is segmented into three components: a *feature store*, a *graph store*, and a *graph sampler*, *cf.* Figure 1. The data loader calls the graph sampler with a set of seed nodes, which performs graph sampling on the graph store and returns a set of subgraph structures. The data loader subsequently requests the features of sampled nodes and edges from the feature store, and joins the features with the sampled subgraph to construct a PyG mini-batch object that can be directly used within its neural framework. Users that define custom feature handling are only required to specify the implementation of the `get` operation on their feature backend, and users that define custom graph handling are required to specify how sampling is performed against their graph representation. As a result, while the graph and feature stores can be independently partitioned, replicated, and stored in optimized formats, the training loop can operate oblivious to these details. These abstractions are also foundational to the in-memory storage formats used in PyG. Specifically, both `Data` and `HeteroData` classes in PyG inherit from the `FeatureStore` and `GraphStore` interfaces, providing a unified mechanism for retrieving mini-batches from any type of data storage throughout the whole code base.

Efficient Subgraph Sampling. Subgraph sampling [1, 12, 14, 18, 30, 37, 42, 44, 59, 102, 103, 106] is a common technique used to

scale graph learning to large graphs. Instead of aggregating messages from all neighbors, only a subset of neighbors up to k-hops are sampled for each node of interest. This reduces memory and computational cost, making mini-batch training feasible even on billion-scale graphs.

Despite its advantages, subgraph sampling can be inefficient if implemented naively. Pure Python-based implementations suffer from interpreter overhead and are constrained by the Global Interpreter Lock (GIL). To mitigate these issues, PyG introduces a high-performance custom C++ homogeneous and heterogeneous subgraph sampling pipeline that supports multi-threading both across edge types and across data loader workers. The underlying implementation is highly flexible to support different needs: Users can seamlessly move between disjoint or intersecting subgraphs within a mini-batch, and can tune the output to be either directional or bi-directional (*e.g.*, in order to implement deep GNNs on shallow subgraphs [1, 102]).

Unlike other GNN libraries that return layer-wise 1-hop subgraphs for neighbor sampling [87], PyG produces a single multi-hop subgraph. This design enables seamless transitions between full-batch and mini-batch training, supports interchangeable graph sampling strategies, and promotes a clean separation between model architecture and data loading, making the overall workflow more modular and flexible.

Trim. However, in some scenarios, this flexibility comes at the cost of performance, as the model cannot exploit special characteristics of the underlying data loading routine. One such limitation is that a GNN trained on a Breadth First Search (BFS)-generated subgraph learns representations for *all* nodes at *all* depths of the network, although nodes sampled in later hops do not contribute to the representations of seed nodes in later GNN layers anymore, thus performing redundant computation. To maximize efficiency, we introduce a layer-wise pruning mechanism which progressively trims the adjacency matrix of the returned subgraph. This progressive trimming is done by simply slicing the adjacency and feature matrices according to the BFS ordering on-the-fly, ensuring zero-copying throughout the process. This approach combined with model compilation leads to a 4-5x speed up, *cf.* Table 1.

Temporal Subgraph Sampling. PyG 2.0 supports both *temporal* homogeneous and heterogeneous graphs as part of its subsampling routines [27, 90]. Temporal subgraph sampling in PyG enables seamless traversal of dynamic graphs over time, allowing to extract (sub)graph snapshots at any point in time.

Given a seed node v and a seed timestamp t, the resulting k-hop subgraph $G_k^{\leq t}[v]$ around node v is constructed such that all included nodes and edges respect temporal constraints—specifically, they must have appeared at or before timestamp t. This ensures the subgraph contains no future information, thereby preventing temporal leakage. For node and edge types lacking timestamps (*e.g.*, institutions or locations), sampling is performed without applying temporal constraints.

A variety of temporal sampling strategies are supported, including uniform sampling, sampling the most recent k nodes or edges, and annealing-based strategies that gradually bias sampling toward more recent elements. Within each mini-batch, the sampled

subgraphs are guaranteed to be disjoint, permitting different seed timestamps across samples while maintaining temporal consistency.

cuGraph Integration. Based on our `FeatureStore` and `GraphStore` abstractions, we enabled end-to-end GPU-accelerated PyG workflows via cuGraph integration. The cuGraph<>PyG extension, part of the NVIDIA RAPIDS [64] framework, is built upon *cuGraph* [63] for GPU-accelerated graph analytics and sampling, and *WholeGraph* [95, 96] for GPU-accelerated distributed tensor and embedding storage. This enables 2x-8x data loading speedups with minimal code change, even for single-GPU workflows, and can be easily extended to multi-node multi-GPU setups. All workflows benefit from a fast bulk sampling process on the GPU, which generates samples for as many batches as possible in parallel. Then, during the feature fetching stage, WholeGraph allows features to be distributed across workers efficiently, which minimizes synchronization overhead, reduces memory transfers, and removes redundant data copies. Through cuGraph<>PyG, it is possible to achieve linear scaling when stacking additional GPUs.

2.4 Explainability

Explainability of machine learning models has become increasingly important for a range of reasons, including trust, regulatory compliance, security, and ease of debugging. Unlike traditional machine learning models, GNNs operate over irregular and relational data structures, making their decision-making processes inherently more difficult to interpret—particularly when it comes to understanding both feature and structural influence.

PyG 2.0 provides comprehensive support for explaining (heterogeneous) GNNs through its universal `Explainer` interface (*cf.* Figure 2). The `Explainer` class acts as a bridge between user-defined GNNs, explanation algorithms, and graph data, to generate attributions that signify the importance of nodes, edges, and features in the model's decision-making process. Formally, given a GNN : $\mathcal{G} \to \mathcal{Y}$, mapping an input graph to a prediction, we seek to find attributions $\mathbf{A}_{\mathcal{V}} \in \mathbb{R}^{|\mathcal{V}| \times F}$ and $\mathbf{a}_{\mathcal{E}} \in \mathbb{R}^{|\mathcal{E}|}$ that identify the contribution of an individual input feature in $\mathbf{H}^{(0)}$ and $\mathcal{E}$, respectively.

To generate structural explanations $\mathbf{a}_{\mathcal{E}}$ of non-differentiable inputs $\mathcal{E}$, the `Explainer` module temporarily alters the internal message passing process of Eq. (1) in PyG GNNs by enabling customization of messages through a callback mechanism $c : \mathbb{R}^{|\mathcal{E}| \times F} \to \mathbb{R}^{|\mathcal{E}| \times F}$, *i.e.*

$$\mathbf{h}_v^{(\ell+1)} = f\left(\mathbf{h}_v^{(\ell)}, \left\{\!\!\left\{ c\left(g\left(\mathbf{h}_w^{(\ell)}, \mathbf{e}_{(w,v)}, \mathbf{h}_v^{(\ell)} \right) \right) \mid w \in \mathcal{N}(v) \right\}\!\!\right\} \right).$$

This callback allows, *e.g.*, to introduce perturbations, to apply edge-level masks that weight incoming messages, or to capture internal attention coefficients. Afterwards, the explanation algorithm assesses how these modifications affect the model's predictions or align with ground-truth data [3]. Such callback mechanism is applicable both in homogeneous and heterogeneous GNNs. In explanation mode, message passing falls back to edge-level materialization (c.f. Sec. 2.2) in order to uniformly inject c across all edges.

With this modular design, researchers only need to focus on the real challenge of building new and improved explainer algorithms, while the data flow, visualizations and evaluation protocols [2, 3] (*e.g.*, fidelity or unfaithfulness) are handled by the PyG framework.
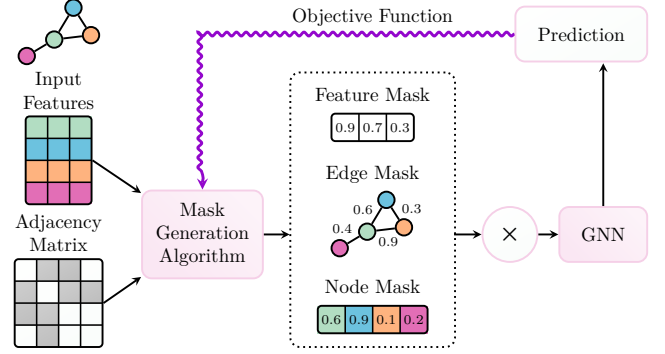

Figure 2: Exemplary illustration of a GNN explainer in PyG: The explainer generates node-level and edge-level masks, which are multiplied within the GNN to weigh node features and message passing edges. The masks are optimized via an objective function to preserve only necessary information.

Captum Integration. While PyG supports a variety of proposed graph-specific explainer modules [58, 75, 98], it also provides a direct connection to Captum [51], a general-purpose explainability library for PyTorch. Captum offers a wide range of out-of-the-box explainers, such as saliency [80], integrated gradients [82], guided backpropagation [81], or deconvolution [101]. While Captum is effective for various data modalities like vision and text, its direct application to GNNs presents challenges due to non-differentiable inputs $\mathcal{E}$. As a consequence, our `CaptumExplainer` module builds a wrapper around any (heterogeneous) PyG GNN such that only the node features and an edge-level soft mask (initialized with ones) are required as input arguments. Internally, the edge-level soft mask is then attached to reweigh messages in every GNN layer via the callback mechanism c. This effectively makes all inputs to the GNN differentiable, which can now be utilized by Captum to explain both feature information and structural properties via its large set of gradient-based explainer modules.

3 Applications: PyG 2.0 in Action

We now provide an overview of applications powered by PyG, with a special focus on Relational Deep Learning (Sec. 3.1) and integration in Large Language Models (Sec. 3.2). Last, we provide a wider overview of further applications and the ecosystem (Sec. 3.3).

3.1 Relational Deep Learning

PyG's support for heterogeneous temporal graphs enables its use for *Relational Deep Learning (RDL)* [27], offering a modern deep learning alternative to traditional feature-based approaches for learning on raw relational databases. In RDL, relational data is represented as a graph, where each entity denotes a node, and the primary-foreign key links between entities define the edges.

PyG supports the full end-to-end RDL blueprint, which covers **(1)** handling of *multi-modal* data, **(2)** querying historical subgraphs based on the contents of a *training table*, and **(3)** *recommender system* support. For handling multi-modal data, we integrated support for PyTorch Frame [41] into feature fetching capabilities of our

`FeatureStore`. That is, we allow nodes to hold multi-modal data according to their semantic type (*e.g.*, numericals, (multi-)categoricals, timestamps, free text), stored in a `TensorFrame` [41]. Afterwards, we can combine existing table-encoding algorithms [4, 13, 15, 35, 45] from deep tabular learning jointly with GNN message passing algorithms for cross-table information exchange (*cf.* Figure 3).

Furthermore, RDL requires flexibility as part of data loading routines, where seed nodes, their timestamps, and corresponding labels are defined externally via a training table. To accommodate this, PyG 2.0 enables subgraph samplers to iterate over externally specified seed nodes and timestamps, extracting subgraphs centered around the appropriate node types. Ground-truth labels and other training table metadata can be dynamically attached to these subgraphs through the concept of `transforms`, which allow customization into the feature fetching pipeline.

Finally, PyG 2.0 offers full support for GNN-based recommender systems, including efficient *Maximum Inner Product Search (MIPS)* via the FAISS library [24], as well as mini-batch-compatible retrieval metrics (*e.g.*, `map@k` or `ndcg@k`), implemented according to `torchmetrics` [23] standards. This elevates link prediction GNNs beyond the conventional binary classification paradigm—restricted to pre-defined candidate pairs—into realistic recommendation scenarios where candidate items are not known a priori.

3.2 Integration in Large Language Models

PyG contributes to the LLM domain in two different ways: **(1)** it provides examples how to utilize LLM embeddings as part of text-attributed graphs [16, 39, 88] in graph learning, and **(2)** by supporting various techniques for *Retrieval Augmented Generation (RAG)* [54], as detailed in the following.

RAG enables LLMs to incorporate document databases as contextual knowledge sources. To capture the underlying structure of these databases, models such as GNNs and Graph Transformers are used to enhance the LLMs' ability to reason over relational and topological information—a technique commonly referred to as *GraphRAG* [25].

GraphRAG starts with a natural language query, which is used to retrieve a relevant contextual subgraph from a larger knowledge graph database. This subgraph is then encoded using a GNN, and the resulting node embeddings are aggregated and projected into the LLM's embedding space. PyG supports this retrieval workflow through extensions to its `FeatureStore` and `GraphStore` abstractions. The interface is fully customizable and can be adapted to domain-specific retrieval strategies, *cf.* Figure 4.

The full GNN+LLM generator pipeline in PyG is enabled via the `G-Retriever` model [40] which allows any combination of a PyG GNN with a HuggingFace LLM [91]. Notably, the addition of GNNs provides a 2x increase in accuracy over pure LLM baselines [78], improving from 16% (LLM-based Agentic RAG) to 32% (GNN+LLM-based Graph RAG) accuracy.

Furthermore, PyG provides the `TXT2KG` class, an easy-to-use interface to convert unstructured text datasets into a knowledge graph via parsing and prompt engineering.
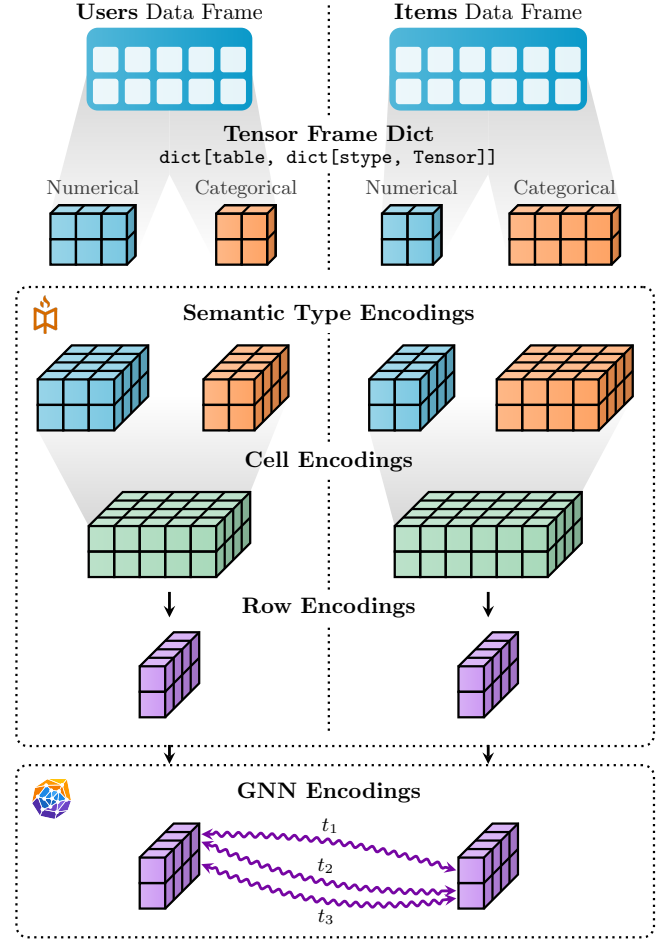

Figure 3: End-to-end Relational Deep Learning on multi-modal and multi-table data with PyTorch Frame and PyG: Every row in each table is encoded individually using tabular deep learning algorithms. Afterwards, message passing GNNs can be applied to exchange cross-table information.

3.3 Others Application Areas

In recent years, PyG has been applied in a large variety of further application areas. In the following, we highlight a few important examples that showcase the wide range of applicability.

Chemistry. Graph neural networks and efficient GPU implementations such as PyG have been largely successful in chemistry [33, 67]. PyG has been used for drug discovery by combining GNN models with chemical foundation models [5]. Research infrastructure for material discovery with GNNs has been built on top of PyG [32] opening applications of GNNs for surface material property prediction [60].

Large Spatial Graphs. Due to a large amount of data on connected nodes, PyG enables data-driven weather forecasting, as demonstrated by research in probabilistic weather forecasting [65, 66]. It was adopted by researchers from the European Centre for Medium-Range Weather Forecasts (ECMWF) to built a data-driven weather
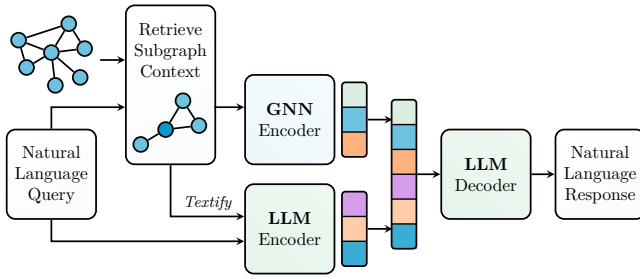
Figure 4: The general GraphRAG pipeline in PyG: A natural language query is used to retrieve relevant contextual subgraphs from a larger knowledge graph, which are then encoded via a GNN. The resulting node embeddings are used to enhance the conventional LLM encoder<>decoder flow.

forecasting system [52]. Similarly, PyG was applied to analyze and predict behavior in traffic scenarios [46, 50].

Optimization. More recently, GNNs have become a dominant paradigm to solve combinatorial optimization problems [11, 76]. Multiple solvers have been developed on top of PyG [47, 69]. The field has been identified as one of the future fields with much potential GNNs development [8].

Social Network Analysis. PyG has been used for a wide arrange of social network analysis tasks, such as bot detection [26, 97], community detection [20], and fake news detection [61].

Computer Vision. In the area of computer vision, PyG has been applied to process irregularly structured data, such as unstructured point clouds [53, 105], meshes [31], and scene graphs [104]. It found application to solve tasks like matching [29], autonomous driving [100], and grasp analysis [9].

Ecosystem. PyG sparked the creation of a vibrant ecosystem of open-source software built on top and around it. Examples include Quiver [84], a library for distributed training, AutoGL [36], an AutoML framework, DIG [57] with higher level extensions, and Pytorch Geometric Temporal [73] for temporal graphs. Additionally, PyGOD [56] adds functionality for outlier and anomaly detection, FedGraphNN [38] provides federated learning capabilities, and Pytorch Frame [41] adds encoders for tabular data. In addition to functionality, benchmarks, such as Relbench [72] for relational data and temporal graphs [43] have completed the package.

4 Conclusion

PyG 2.0 represents a significant advancement in graph learning frameworks, offering scalable solutions for real-world applications while maintaining ease of use and flexibility. We presented advances in three different categories, scalable graph infrastructure, the neural framework, and post-processing techniques such as explainability, showcasing the highly modular framework design. PyG has been applied in a wide range of applied fields, including the very recent areas of relational deep learning and RAG systems in large language models, in which we expect further significant developments in the near future.

References

[1] R. Addanki, P. W. Battaglia, D. Budden, A. Deac, J. Godwin, T. Keck, W. L. Sibon Li, A. Sanchez-Gonzalez, J. Stott, S. Thakoor, and P. Veličković. 2021. Large-scale Graph Representation Learning with Very Deep GNNs and Self-supervision. *CoRR* abs/2107.09422 (2021).

[2] Chirag Agarwal, Owen Queen, Himabindu Lakkaraju, and Marinka Zitnik. 2023. Evaluating Explainability for Graph Neural Networks. *CoRR* abs/2208.09339 (2023).

[3] Kenza Amara, Zhitao Ying, Zitao Zhang, Zhichao Han, Yang Zhao, Yinan Shan, Ulrik Brandes, Sebastian Schemm, and Ce Zhang. 2022. GraphFramEx: Towards Systematic Evaluation of Explainability Methods for Graph Neural Networks. In *LOG*.

[4] Sercan Ö Arik and Tomas Pfister. 2021. TabNet: Attentive interpretable tabular learning. In *AAAI*.

[5] Kenneth Atz, Leandro Cotos, Clemens Isert, Maria Håkansson, Dorota Focht, Mattis Hilleke, David F. Nippa, Michael Iff, Jann Ledergerber, Carl C. G. Schiebroek, Valentina Romeo, Jan A. Hiss, Daniel Merk, Petra Schneider, Bernd Kuhn, Uwe Grether, and Gisbert Schneider. 2024. Prospective De Novo Drug Design with Deep Interactome Learning. *Nature Communications* (2024).

[6] Sergey Bartunov, Fabian B. Fuchs, and Timothy Lillicrap. 2022. Equilibrium Aggregation: Encoding Sets via Optimization. In *UAI*.

[7] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. 2018. Relational inductive biases, deep learning, and graph networks. *CoRR* abs/1806.01261 (2018).

[8] Maya Bechler-Speicher, Ben Finkelshtein, Fabrizio Frasca, Luis Müller, Jan Tönshoff, Antoine Siraudin, Viktor Zaverkin, Michael M. Bronstein, Mathias Niepert, Bryan Perozzi, Mikhail Galkin, and Christopher Morris. 2025. Position: Graph Learning Will Lose Relevance Due To Poor Benchmarks. In *ICML*.

[9] Samarth Brahmbhatt, Chengcheng Tang, Christopher D. Twigg, Charles C. Kemp, and James Hays. 2020. ContactPose: A Dataset of Grasps with Object Contact and Hand Pose. In *ECCV*.

[10] Michael Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. *CoRR* abs/2104.13478 (2021).

[11] Quentin Cappart, Didier Chételat, Elias B. Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. 2023. Combinatorial Optimization and Reasoning with Graph Neural Networks. *JMLR* (2023).

[12] J. Chen, T. Ma, and C. Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *ICLR*.

[13] Jintai Chen, Jiahuan Yan, Danny Ziyi Chen, and Jian Wu. 2023. ExcelFormer: A Neural Network Surpassing GBDTs on Tabular Data. *CoRR* abs/2301.02819 (2023).

[14] J. Chen, J. Zhu, and L. Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *ICML*.

[15] Kuan-Yu Chen, Ping-Han Chiang, Hsin-Rung Chou, Ting-Wei Chen, and Tien-Hao Chang. 2023. Trompt: Towards a Better Deep Neural Network for Tabular Data. In *ICML*.

[16] Z. Chen, H. Mao, J. Liu, Y. Song, B. Li, W. Jin, B. Fatemi, A. Tsitsulin, B. Perozzi, H. Liu, and J. Tang. 2024. Text-space Graph Foundation Models: Comprehensive Benchmarks and New Insights. *CoRR* 2406.10727 (2024).

[17] Zhiru Zhang Chenhui Deng, Zichao Yue. 2024. Polynormer: Polynomial-Expressive Graph Transformer in Linear Time. In *ICLR*.

[18] W. L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C. J. Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *KDD*.

[19] G. Corso, L. Cavalleri, D. Beaini, P. Liò, and P. Veličković. 2020. Principal Neighbourhood Aggregation for Graph Nets. In *NeurIPS*.

[20] Aurélio Ribeiro Costa and Célia Ghedini Ralha. 2023. AC2CD: An actor–critic architecture for community detection in dynamic social networks. *Knowledge-Based Systems* (2023).

[21] M. Defferrard, X. Bresson, and P. Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*.

[22] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research. *IEEE Signal Processing Magazine* (2012).

[23] Nicki Skafte Detlefsen, Jiri Borovec, Justus Schock, Ananya Harsh, Teddy Koker, Luca Di Liello, Daniel Stancl, Changsheng Quan, Maxim Grechkin, and William Falcon. 2022. TorchMetrics: Machine Learning Metrics for PyTorch.

[24] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The FAISS Library. *CoRR* abs/2401.08281 (2024).

[25] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and

Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-focused Summarization. *CoRR* (2024).

[26] Shangbin Feng, Zhaoxuan Tan, Herun Wan, Ningnan Wang, Zilong Chen, Binchi Zhang, Qinghua Zheng, Wenqian Zhang, Zhenyu Lei, Shujie Yang, Xinshun Feng, Qingyue Zhang, Hongrui Wang, Yuhan Liu, Yuyang Bai, Heng Wang, Zijian Cai, Yanbo Wang, Lijing Zheng, Zihan Ma, Jundong Li, and Minnan Luo. 2022. TwiBot-22: Towards Graph-Based Twitter Bot Detection. In *NeurIPS*.

[27] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. 2024. Position: Relational Deep Learning: Graph Representation Learning on Relational Databases. In *ICML*.

[28] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *ICLR (RLGM Workshop)* (2019).

[29] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege. 2020. Deep Graph Matching Consensus. In *ICLR*.

[30] M. Fey, J. E. Lenssen, F. Weichert, and J. Leskovec. 2021. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *ICML*.

[31] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. 2018. SplineCNN: Fast Geometric Deep Learning With Continuous B-Spline Kernels. In *CVPR*.

[32] Victor Fung, Jiaxin Zhang, Eric Juarez, and Bobby G. Sumpter. 2021. Benchmarking Graph Neural Networks for Materials Chemistry. *npj Computational Materials* (2021).

[33] Ziqi Gao, Chenran Jiang, Jiawen Zhang, Xiaosen Jiang, Lanqing Li, Peilin Zhao, Huanming Yang, Yong Huang, and Jia Li. 2023. Hierarchical Graph Learning for Protein–Protein Interaction. *Nature Communications* (2023).

[34] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *ICML*.

[35] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. 2021. Revisiting Deep Learning Models for Tabular Data. In *NeurIPS*.

[36] Chaoyu Guan, Ziwei Zhang, Haoyang Li, Heng Chang, Zeyang Zhang, Yijian Qin, Jiyan Jiang, Xin Wang, and Wenwu Zhu. 2021. AutoGL: A Library for Automated Graph Learning. In *ICLR (GTRL Workshop)*.

[37] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NeurIPS*.

[38] Chaoyang He, Songze Li, Jinhyun So, Xiao Zeng, Mi Zhang, Hongyi Wang, Xiaoyang Wang, Praneeth Vepakomma, Abhishek Singh, Hang Qiu, Xinghua Zhu, Jianzong Wang, Li Shen, Peilin Zhao, Yan Kang, Yang Liu, Ramesh Raskar, Qiang Yang, Murali Annavaram, and Salman Avestimehr. 2020. FedML: A Research Library and Benchmark for Federated Machine Learning. *CoRR* abs/2007.13518 (2020).

[39] X. He, X. Bresson, T. Laurent, A. Perold, Y. LeCun, and B. Hooi. 2024. Harnessing Explanations: LLM-to-LM Interpreter for Enhanced Text-Attributed Graph Representation Learning. In *ICLR*.

[40] Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering. *CoRR* abs/2402.07630 (2024).

[41] Weihua Hu, Yiwen Yuan, Zecheng Zhang, Akihiro Nitta, Kaidi Cao, Vid Kocijan, Jinu Sunil, Jure Leskovec, and Matthias Fey. 2024. PyTorch Frame: A Modular Framework for Multi-Modal Tabular Learning. *CoRR* abs/2404.00776 (2024).

[42] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous Graph Transformer. In *WWW*.

[43] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2023. Temporal Graph Benchmark for Machine Learning on Temporal Graphs. In *NeurIPS*.

[44] W. Huang, T. Zhang, Y. Rong, and J. Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *NeurIPS*.

[45] Xin Huang, Ashish Khetan, Milan Cvitkovic, and Zohar Karnin. 2020. TabTransformer: Tabular Data Modeling using Contextual Embeddings. *CoRR* abs/2012.06678 (2020).

[46] Maria Huegle, Gabriel Kalweit, Moritz Werling, and Joschka Boedecker. 2020. Dynamic Interaction-Aware Scene Understanding for Reinforcement Learning in Autonomous Driving. In *ICRA*.

[47] Nikolaos Karalias and Andreas Loukas. 2020. Erdos Goes Neural: an Unsupervised Learning Framework for Combinatorial Optimization on Graphs. In *NeurIPS*.

[48] Jinwoo Kim, Tien Dat Nguyen, Seonwoo Min, Sungjun Cho, Moontae Lee, Honglak Lee, and Seunghoon Hong. 2022. Pure Transformers are Powerful Graph Learners. In *NeurIPS*.

[49] T. N. Kipf and M. Welling. 2017. Semi-supervised Classification with Graph Convolutional Networks. In *ICLR*.

[50] Marvin Klimke, Benjamin Völz, and Michael Buchholz. 2022. Cooperative Behavior Planning for Automated Driving Using Graph Neural Networks. In *IV*.

[51] Narine Kokhlikyan, Vivek Miglani, Miguel Martin, Edward Wang, Bilal Alsallakh, Jonathan Reynolds, Alexander Melnikov, Natalia Kliushkina, Carlos Araya, Siqi

[52] Yan, and Orion Reblitz-Richardson. 2020. Captum: A Unified and Generic Model Interpretability Library for PyTorch. *CoRR* abs/2009.07896 (2020).

[52] Simon Lang, Mihai Alexe, Matthew Chantry, Jesper Dramsch, Florian Pinault, Baudouin Raoult, Mariana Clare, Christian Lessig, Michael Maier-Gerber, Linus Magnusson, Zied Ben Bouallègue, Ana Nemesio, Peter Düben, Andrew Brown, Florian Pappenberger, and Florence Rabier. 2024. AIFS - ECMWF's Data-driven Forecasting System. *CoRR* abs/2406.01465 (2024).

[53] Jan Eric Lenssen, Christian Osendorfer, and Jonathan Masci. 2020. Deep Iterative Surface Normal Estimation. In *CVPR*.

[54] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *CoRR* abs/2005.11401 (2020).

[55] Guohao Li, Chenxin Xiong, Ali Thabet, and Bernard Ghanem. 2020. DeeperGCN: All You Need to Train Deeper GCNs. *CoRR* abs/2006.07739 (2020).

[56] Kay Liu, Yingtong Dou, Xueying Ding, Xiyang Hu, Ruitong Zhang, Hao Peng, Lichao Sun, and Philip S. Yu. 2024. PyGOD: A Python Library for Graph Outlier Detection. *JMLR* (2024).

[57] Meng Liu, Youzhi Luo, Limei Wang, Yaochen Xie, Hao Yuan, Shurui Gui, Haiyang Yu, Zhao Xu, Jingtun Zhang, Yi Liu, Keqiang Yan, Haoran Liu, Cong Fu, Bora M Oztekin, Xuan Zhang, and Shuiwang Ji. 2021. DIG: A Turnkey Library for Diving into Graph Deep Learning Research. *JMLR* (2021).

[58] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized Explainer for Graph Neural Network. *CoRR* abs/2011.04573 (2020).

[59] E. Markowitz, K. Balasubramanian, M. Mirtaheri, S. Abu-El-Haija, B. Perozzi, G. Ver Steeg, and A. Galstyan. 2021. Graph Traversal with Tensor Functionals: A Meta-Algorithm for Scalable Learning. In *ICLR*.

[60] Marco Maurizi, Chao Gao, and Filippo Berto. 2022. Predicting Stress, Strain and Deformation Fields in Materials and Structures with Graph Neural Networks. *Scientific Reports* (2022).

[61] Dimitrios Michail, Nikos Kanakaris, and Iraklis Varlamis. 2022. Detection of Fake News Campaigns using Graph Convolutional Networks. *IJIM Data Insights* (2022).

[62] Xiaoyu Mo, Yang Xing, and Chen Lv. 2021. Heterogeneous Edge-Enhanced Graph Attention Network For Multi-Agent Trajectory Prediction. *CoRR* abs/2106.07161 (2021).

[63] NVIDIA Corporation. 2025. RAPIDS cuGraph. https://docs.rapids.ai/api/cugraph.

[64] NVIDIA Corporation. 2025. RAPIDS: GPU Accelerated Data Science. https://rapids.ai.

[65] Joel Oskarsson, Tomas Landelius, Marc Peter Deisenroth, and Fredrik Lindsten. 2024. Probabilistic Weather Forecasting with Hierarchical Graph Neural Networks. In *NeurIPS*.

[66] Joel Oskarsson, Tomas Landelius, and Fredrik Lindsten. 2023. Graph-based Neural Weather Prediction for Limited Area Modeling. In *NeurIPS (Workshop on Tackling Climate Change with Machine Learning)*.

[67] Mohit Pandey, Michael Fernandez, Francesco Gentile, Olexandr Isayev, Alexander Tropsha, Abraham C. Stern, and Artem Cherkasov. 2022. The Transformational Role of GPU Computing and Deep Learning in Drug Discovery. *Nature Machine Intelligence* (2022).

[68] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*.

[69] Ruizhong Qiu, Zhiqing Sun, and Yiming Yang. 2022. DIMES: A Differentiable Meta Solver for Combinatorial Optimization Problems. In *NeurIPS*.

[70] L. Rampášek, M. Galkin, V. Prakash, A. T. Luu, G. Wold, and D. Beaini. 2022. Recipe for a General, Powerful, Scalable Graph Transformer. In *NeurIPS*.

[71] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2021. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. *CoRR* abs/2112.08429 (2021).

[72] Joshua Robinson, Rishabh Ranjan, Weihua Hu, Kexin Huang, Jiaqi Han, Alejandro Dobles, Matthias Fey, Jan E. Lenssen, Yiwen Yuan, Zecheng Zhang, Xinwei He, and Jure Leskovec. 2024. RelBench: A Benchmark for Deep Learning on Relational Databases.

[73] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzman Lopez, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *CIKM*.

[74] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web*.

[75] Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. 2021. Interpreting Graph Neural Networks for NLP with Differentiable Edge Masking. In *ICLR*.

[76] Martin J. A. Schuetz, John Kyle Brubaker, and Helmut G. Katzgraber. 2021. Combinatorial Optimization with Physics-inspired Graph Neural Networks.

Nature Machine Intelligence (2021).

[77] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Mag.* (2008).

[78] Brian Shi, Alfred Clemedtson, Zach Blumenfeld, and Rishi Puri. 2025. *Boosting Q&A Accuracy with GraphRAG Using PyG and Graph Databases*. https://developer.nvidia.com/blog/boosting-qa-accuracy-with-graphrag-using-pyg-and-graph-databases

[79] Hamed Shirzad, Ameya Velingker, Balaji Venkatachalam, Danica J Sutherland, and Ali Kemal Sinop. 2023. Exphormer: Sparse Transformers for Graphs. In *ICML*.

[80] K. Simonyan, A. Vedaldi, and A. Zisserman. 2013. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *CoRR* 1312.6034 (2013).

[81] J.T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. 2015. Striving for Simplicity: The All Convolutional Net. In *ICLR (Workshop Track)*.

[82] M. Sundararajan, A. Taly, and Q. Yan. 2024. Axiomatic Attribution for Deep Networks. In *ICML*.

[83] Shyam A. Tailor, Felix L. Opolka, Pietro Liò, and Nicholas D. Lane. 2022. Do We Need Anisotropic Graph Neural Networks?. In *ICLR*.

[84] Zeyuan Tan, Xiulong Yuan, Congjie He, Man-Kit Sit, Guo Li, Xiaoze Liu, Baole Ai, Kai Zeng, Peter Pietzuch, and Luo Mai. 2023. Quiver: Supporting GPUs for Low-Latency, High-Throughput GNN Serving with Workload Awareness. *CoRR* abs/2305.10863 (2023).

[85] Vijay Thakkar, Pradeep Ramani, Cris Cecka, Aniket Shivam, Honghao Lu, Ethan Yan, Jack Kosaian, Mark Hoemmen, Haicheng Wu, Andrew Kerr, Matt Nicely, Duane Merrill, Dustyn Blasig, Fengqi Qiao, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Manish Gupta. 2023. CUTLASS. https://github.com/NVIDIA/cutlass.

[86] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. 2018. Graph Attention Networks. In *ICLR*.

[87] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *CoRR* abs/1909.01315 (2019).

[88] S. Wang, J. Huang, Z. Chen, Y. Song, W. Tang, H. Mao, W. Fan, H. Liu, X. Liu, D. Yin, and Q. Li. 2025. Graph Machine Learning in the Era of Large Language Models (LLMs). *TIST* (2025).

[89] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous Graph Attention Network. In *WWW*.

[90] Yiwei Wang, Yujun Cai, Yuxuan Liang, Henghui Ding, Changhu Wang, and Bryan Hooi. 2021. Time-Aware Neighbor Sampling for Temporal Graph Networks. *CoRR* abs/2112.09845 (2021).

[91] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace'sTransformers: State-of-the-Art Natural Language Processing. In *ACL*.

[92] Qitian Wu, Wentao Zhao, Chenxiao Yang, Hengrui Zhang, Fan Nie, Haitian Jiang, Yatao Bian, and Junchi Yan. 2024. SGFormer: Simplifying and Empowering Transformers for Large-Graph Representations. *CoRR* abs/2306.10759 (2024).

[93] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *ICLR*.

[94] Carl Yang, Aydın Buluç, and John D Owens. 2018. Design Principles for Sparse Matrix Multiplication on the GPU. In *Euro-PAR*.

[95] Dongxu Yang. 2024. Optimizing Memory and Retrieval for Graph Neural Networks with WholeGraph (Part 1). https://developer.nvidia.com/blog/optimizing-memory-and-retrieval-for-graph-neural-networks-with-wholegraph-part-1.

[96] Dongxu Yang. 2024. Optimizing Memory and Retrieval for Graph Neural Networks with WholeGraph (Part 2). https://developer.nvidia.com/blog/optimizing-memory-and-retrieval-for-graph-neural-networks-with-wholegraph-part-2.

[97] Yingguang Yang, Renyu Yang, Yangyang Li, Kai Cui, Zhiqin Yang, Yue Wang, Jie Xu, and Haiyong Xie. 2023. RoSGAS: Adaptive Social Bot Detection with Reinforced Self-supervised GNN Architecture Search. *ACM Trans. Web* (2023).

[98] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *NeurIPS*.

[99] J. You, R. Ying, and J. Leskovec. 2020. Design Space for Graph Neural Networks. In *NeurIPS*.

[100] Shih-Yuan Yu, Arnav Vaibhav Malawade, Deepan Muthirayan, Pramod P. Khargonekar, and Mohammad Abdullah Al Faruque. 2022. Scene-Graph Augmented Data-Driven Risk Assessment of Autonomous Vehicle Decisions. *IEEE Transactions on Intelligent Transportation Systems* (2022).

[101] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *ECCV*.

[102] Hanqing Zeng, Muhan Zhang, Yinglong Xia, Ajitesh Srivastava, Andrey Malevich, Rajgopal Kannan, Viktor Prasanna, Long Jin, and Ren Chen. 2025. Decoupling the Depth and Scope of Graph Neural Networks. In *NeurIPS*.

[103] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *ICLR*.

[104] Chaoyi Zhang, Jianhui Yu, Yang Song, and Weidong Cai. 2021. Exploiting Edge-Oriented Reasoning for 3D Point-Based Scene Graph Analysis. In *CVPR*.

[105] Yang Zhang, Zixiang Zhou, Philip David, Xiangyu Yue, Zerong Xi, Boqing Gong, and Hassan Foroosh. 2020. PolarNet: An Improved Grid Representation for Online LiDAR Point Clouds Semantic Segmentation. In *CVPR*.

[106] D. Zou, Z. Hu, Y. Wang, S. Jiang, Y. Sun, and Q. Gu. 2019. Layer-Dependent Importance Sampling for Training Deep and Large Graph Convolutional Networks. In *NeurIPS*.