Politecnico di Torino

Politecnico di Torino

Master's degree in Computer Engineering 2024/2025
Graduation Session 10/2025

Graph Neural Networks for Relational Databases Analysis

Relatori:
Paolo Garza
Luca Colomba
Daniele Loiacono

Candidati:
Andrea Mirenda

Acknowledgements

To my parents and my brothers, constant sources of strength and support, who believed in me even when I did not.

To my grandparents, who anchored me in honesty and taught me the simple, enduring value of doing the right thing, of working on my own with perseverance, and of never giving up or complaining.

To my supervisor, Professor Paolo Garza, and to Luca Colomba, whose trust and insight shaped this work, allowing me to explore this fascinating line of research with unwavering support.

Table of Contents

List of Figures

Abstract

Relational databases are the backbone of modern data infrastructure, supporting much of the digital economy. Despite their importance, their rich relational information is often overlooked. Most predictive pipelines, in fact, still flatten relational schemas into a single table, discarding higher-order relational structure, cross-table dependencies, and forcing reliance on costly and fragile feature engineering, sensitive to expert skill.

This thesis embraces a graph native learning alternative: we cast relational schemas into heterogeneous temporal graphs. Each table in the relational schema becomes a node type, rows become nodes and foreign keys become typed edges. Some node types are associated with time attributes, representing the timestamp at which a node appears. Crucially, the graph construction is schema agnostic and automatic: given any relational database, we derive its heterogeneous temporal graph and train a single pipeline for node level regression and classification tasks.

Guided by the goal of efficient and transparent prediction for heterogeneous temporal graphs, this thesis advances the field in three complementary ways: (i) we evaluate self-supervised pre-training strategies tailored to heterogeneous temporal graphs; (ii) we conduct a systematic exploration of graph-model architectures, training regimes, and design choices to surface robust configurations; (iii) we introduce XMetaPath, a self-explainable GNN that aggregates information over a compact set of X meta-paths and provides faithful explanations for each prediction.

On top of this, we conduct a systematic study of meta-path selection and implement three methods that automatically discover meta-paths: greedy based, LLM guided scoring, and a reinforcement learning agent that leverages model feedback to prioritize task relevant relational patterns.

We evaluate these contributions on RelBench, which is a benchmark of realistic multi-table relational datasets with standardized tasks and splits, and we achieve competitive results while introducing a self-explainable model that provides transparent reasoning.

Taken together, our experiments yield two distinct takeaways. First, self-supervised pre-training offers reliable gains in this setting. Second, a self-explainable, meta-path–based model provides transparent rationales while matching, and sometimes surpassing, the predictive power of strong non interpretable baselines.

Chapter 1

Introduction

The widespread adoption of relational databases across domains such as e-commerce, finance, healthcare, and scientific research has led to the accumulation of vast amounts of structured, interlinked data[1][2]. This, in turn, has made relational databases the most prevalent data management model, often serving as the de facto substrate for both operational and analytical workloads across many sectors. This prevalence is not accidental: relational databases offer robustness, scalability, and powerful query languages (e.g., SQL) that enable expressive retrieval across multiple tables[3], these databases are not only widespread, but also semantically rich, encoding both local attributes (columns) and high-level structural constraints (keys, schemas, and timestamps). Relational database management systems (RDBMSs) remain the backbone of enterprise data management. Recent well-known studies across diverse verticals reinforce this centrality of relational data over a huge variery of different fields. Relational datasets are not merely commonplace: they are mission-critical across healthcare, finance, retail, high-energy physics, and a broad variety of domains. Consequently, **any machine-learning method able to exploit native relational structure can expect immediate cross-domain impact**.

These datasets are typically organized in multiple tables connected via primary and foreign key relationships.

Because relational databases often contain both timestamped records and inter-entity links, the predictive tasks they enable tend to be both **temporal** and **relational** in nature. For instance, forecasting future product sales, anticipating customer churn, or modeling patient outcomes over time[4].

Given the central role of relational databases, it is essential to examine how we manage them and assess whether we truly leverage their full potential and relational richness. Today, most machine learning pipelines still operate on a **flattened** view of the data. This process, commonly referred to as **manual propositionalization** or **feature extraction**, typically involves manually joining

multiple tables into a single "flattened" table, a step that **discards structural and relational information**, introduces redundancy, and, critically, demands substantial domain knowledge to perform the flattening correctly[4].

The reason this flattening ritual is so pervasive is that standard ML toolchains expect a single, fixed-width feature matrix. Consequently, practitioners first collapse the database into one 'analysis table' by joining along selected PK–FK paths and aggregating temporal histories into fixed-length summaries. This *flattening-based* strategy has formed the basis of enterprise analytics because the resulting flattened dataset, now reduced to a single table with one row per prediction unit (e.g., one row per customer), is compatible with tabular machine learning models such as decision tree ensembles. Widely-used algorithms like XGBoost[5] and LightGBM[6] are typically employed on these inputs due to their strong empirical performance and robustness across a wide range of use cases.

Enterprises have not merely *adopted* flatten-and-learn pipelines: they have turned them into highly-optimised, industrial assets. Data-engineering teams have invested thousands of engineer-hours in hardening these ETL DAGs against schema drift, automating incremental back-fills, and optimising storage costs. *Any* proposal to abandon flattening therefore seems, at first glance, to throw away years of optimisation and operational know-how, a clear step backward in maturity.

Yet this very success hides a ceiling: all this optimization comes at the cost of **discarding much of the relational structure**[7], introducing approximations through aggregation, and embedding strong **inductive biases** via the manual choice of features. Moreover, the feature engineering phase often dominates the total development time in real-world ML pipelines, and is highly sensitive to domain expertise and historical knowledge about the dataset[8].

This **flattening** process manually joins multiple tables, aggregates information over time, and designs handcrafted features that aim to capture key relational and temporal dependencies. For example, a customer churn prediction model may include features such as the number of purchases in the past 30 days, the average order value, or the time since last interaction. These are all features that are not directly present in any single table, but computed by navigating the underlying foreign-key relationships and temporal columns. These operations are often implemented through complex SQL queries, data pipelines, or feature engineering scripts maintained over time. This process is not only time-consuming and resource-intensive, but also inherently brittle and difficult to reproduce. The final outcome often depends on a series of ad-hoc decisions made by the domain expert, which may vary from one execution to another. As a result, the quality and consistency of the extracted features are **subject to human bias**, intuition, and domain-specific knowledge, making the process highly sensitive to individual interpretation and prone to variability across different runs.

Even a sophisticated feature store invariably represents an *implicit hypothesis*

about which interactions matter, because every engineered column collapses one or more joins and time windows into a single scalar. Such design choices are well suited to capturing first-order statistics, such as "total spend in the last 30 days," "mean rating per user," and so on, but they struggle to encode *higher-order dependencies*: the chained influence of a friend's purchase on a user's subsequent review, the subtle seasonality that modulates demand *only* when a specific marketing campaign is active, or the multi-step referral paths that tie communities of customers together. These patterns emerge from **multi-hop, context-dependent relations** that span several tables and timestamps; once the schema is flattened, their connective tissue is irrevocably lost. As a result, the engineered view can look perfectly "complete" while still **omitting the very relational signal that drives the target variable**. Any downstream model is then forced to learn from a depiction of the world that is both narrower and more deterministic than the data truly are, an invisible ceiling on predictive performance.

Another important drawback of this strategy is that a single denormalised table can be *orders of magnitude larger* than the base tables from which it is derived, because every dimension tuple is replicated for each matching fact row, inflating both storage and compute resources. Furthermore, flattened pipelines hard-code column positions and names; when upstream schemas evolve (columns added, types changed), the join logic and feature mappings break.

In summary, while this flattened-data paradigm has historically enabled predictive modeling over relational databases, it relies heavily on human effort and domain heuristics, limiting scalability, reusability, and automation.

At this point it is worth pausing to ask a simple but uncomfortable question: **is flattening truly the only viable route for learning over relational databases?** Must we always compress a web of entities connected by primary and foreign key constraints into a single rectangular view, accepting the loss of structure as a necessary cost? Or could there be an **automatic** and immediate representation that **preserves the native semantics and link structure**, one that lets models **reason with relations** rather than in spite of them, and that carries multiplicities, directions, and cardinalities into the learning pipeline without hand crafted shortcuts?

An innovative answer is to represent the database itself as a **heterogeneous temporal graph**. In this view, each table becomes a node type, each row becomes a node, and each primary to foreign key reference induces a typed edge that connects the corresponding nodes. Attributes live as node or edge features, temporal information can be attached where it belongs, and the schema's constraints appear not as obstacles to be flattened away but as first class signals that guide inference. The richness that was previously squeezed into a single table is now expressed directly in the topology and labels of the graph, so that learning can exploit the very structure that the relational model was designed to capture.

Building on this representation, Relational Deep Learning (RDL)[9] advances beyond manual feature engineering and tabular modeling by using Graph Neural Networks (GNNs)[4] as the core predictive model over heterogeneous graph views of relational databases, where nodes represent entities from different tables and edges capture typed relationships defined by the database schema. These models **bypass the need for flattening** or handcrafted aggregation by directly leveraging the relational structure, enabling end-to-end learning from multi-table data in its native form. By treating relational data as a graph with multiple node and edge types, GNNs can effectively capture multi-hop dependencies, semantic hierarchies, and relational patterns that are often diluted or lost in traditional tabular representations.

More generally, Graph Neural Networks (GNNs) [10] provide a framework for learning representations over graph-structured data. Unlike standard neural networks that assume inputs to be vectors or sequences, GNNs are designed to operate on arbitrary graphs, where both nodes and edges may carry features and vary in size or connectivity. In its typical form, a GNN layer updates each node's representation by combining its current features with aggregated messages from its direct neighbors, this mechanism is often referred to as *message passing* [11]. This architecture allows GNNs to model local structural patterns and context-dependent node semantics. Over multiple layers, GNNs can capture broader, multi-hop interactions within the graph.

Having established the context, we now outline the structure of this work. Chapter 2 surveys related work on tabular models, graph neural networks, node connectivity, popular architectures, explainable graph neural networks with a focus on meta path based methods, and the RelBench[4] end to end modeling pipeline. Chapter 3 presents the experimental study. It compares backbones including a heterogeneous graph attention network and a heterogeneous temporal Graphormer, develops three in domain self supervised pretraining strategies (masked attribute prediction, a variational graph autoencoder, and relationally aware edge dropout), introduces XMetaPath with its single hop MetaPathGNNLayer, and studies meta path selection through greedy validation, a large language model based scorer, and a reinforcement learning policy. Chapter 4 presents the complete empirical evaluation, reporting the full set of experimental results for all proposed models and approaches on RelBench, together with baseline comparisons, and a critical analysis of strengths and limitations. The work closes with the conclusions, future work, and the bibliography.

Chapter 2

Related Works

2.1 The RelBench Benchmark

A central challenge in advancing Relational Deep Learning (RDL) is the lack of standardized benchmarks that faithfully reflect the complexity and variety of real-world relational databases. To address this gap, **RelBench**[4, 9] (Relational Deep Learning Benchmark) was proposed in 2024[1], which contains 7 realistic, large-scale, and diverse relational databases spanning domains including medical, social networks, e-commerce and sport. Each database has multiple predictive tasks defined, each carefully scoped to be both challenging and of domain-specific importance. It provides full support for data downloading, task specification and standardized evaluation in an ML-framework-agnostic manner[4].

RelBench provides a collection of diverse datasets and learning tasks derived directly from real-world relational databases. Rather than relying on synthetic or graph-centric datasets, RelBench is constructed from *multi-table tabular data*, where each dataset includes a rich schema with multiple entities, attributes, timestamps, and foreign-key relationships. The benchmark aims to evaluate the performance of models that can reason over this structure without flattening or manual feature engineering.

Dataset Construction and Schema

Each dataset in RelBench consists of a collection of interrelated tables, connected via primary and foreign key constraints, as commonly found in normalized relational databases. Based on this inherent structure, each dataset can be systematically transformed into a *heterogeneous graph*, where:

[1]It was primarily developed and launched by researchers at Stanford University.

- Each row in an entity table is represented as a distinct **node**.
- A foreign key constraint referencing a primary key in another table is then reinterpreted as a directed, typed edge connecting the corresponding node instances.
- Node attributes are directly derived from the columns of the corresponding table, and may include categorical, numerical, or temporal features.

This graph representation is not handcrafted or dataset-specific, but rather emerges deterministically from the relational schema itself. The resulting graph is heterogeneous, temporal, and attributed, faithfully preserving both the content and the structural relationships defined in the original database. This allows graph-based models to fully exploit the underlying relational context in a principled and reproducible way.

The key goal of RelBench is to enable fair and realistic comparisons between different modeling paradigms:

- **Tabular models**: such as XGBoost and LightGBM [12, 13], applied to flattened data with engineered features.
- **Relational GNNs**: including relational message-passing architectures and heterogeneous graph attention networks [14].
- **Hybrid architectures**: such as transformers with relational context [15], or models that combine column-wise encoders with GNN propagation.

RelBench further reports metrics like ROC-AUC, MAP, and MAE depending on the task, and provides leaderboards for standardized comparisons.

Task Types in RelBench

RelBench defines three primary categories of predictive tasks that reflect common real-world objectives in relational data applications. These task types are:

- **Entity Classification**
  In these tasks, predicting the class label for a single entity (row) is required, where the target variable y is binary, i.e., $y \in \{0,1\}$ (e.g., churn prediction). Evaluation is typically performed using the Area Under the ROC Curve (AUROC).
- **Entity Regression**
  These tasks involve predicting a continuous numeric value for a single entity, i.e., $y \in \mathbb{R}$ (e.g., predicting lifetime value or total amount sold). The performance is usually measured using Mean Absolute Error (MAE).

- **Link-level Prediction (Recommendation)**
  Here, the goal is to predict a relationship between two entities, such as a user-item interaction, rather than properties of a single entity. These tasks are commonly referred to as recommendation tasks. Evaluation is often based on ranking metrics such as Mean Average Precision (MAP).

These category distinctions align well with diverse real-world relational learning scenarios and are central to the structure of RelBench, which comprises 30 tasks across 7 datasets (e.g., entity classification, regression, and recommendation).

RelBench Datasets and Tasks

Below is a comprehensive summary of all datasets in RelBench, including structural statistics and task definitions.

Name	Domain	#Tasks	Tables			Timestamp (year-mon-day)		
			#Tables	#Rows	#Cols	Start	Val	Test
rel-amazon	E-commerce	7	3	15,000,713	15	2008-01-01	2015-10-01	2016-01-01
rel-avito	E-commerce	4	8	20,679,117	42	2015-04-25	2015-05-08	2015-05-14
rel-event	Social	3	5	41,328,337	128	1912-01-01	2012-11-21	2012-11-29
rel-f1	Sports	3	9	74,063	67	1950-05-13	2005-01-01	2010-01-01
rel-hm	E-commerce	3	3	16,664,809	37	2019-09-07	2020-09-07	2020-09-14
rel-stack	Social	5	7	4,247,264	52	2009-02-02	2020-10-01	2021-01-01
rel-trial	Medical	5	15	5,434,924	140	2000-01-01	2020-01-01	2021-01-01
Total		30	51	103,466,370	489	/	/	/

Figure 2.1: Statistics of RELBENCH datasets. Datasets vary significantly in the number of tables, total number of rows, and number of columns. Image taken from Robinson et al.[4].

Task Definitions

While RelBench defines a broad suite of tasks, in this work we focus on those most relevant to our setting. We first provide a concise description of the tasks considered here, and then restrict our empirical study to *node-level prediction*, both node classification and node regression, evaluated on a representative subset of RelBench datasets. This selection balances domain coverage and dataset scale, enabling thorough hyperparameter tuning, and statistical validation under a fixed compute budget; the remaining tasks and larger corpora are left to future work.

rel-avito

- *user-visits*, *user-clicks* (node classification): predict if a user will visit/click more than one ad in the next 4 days. Evaluation metric: **AUROC**.
- *ad-ctr* (node regression): predict the click-through rate for each ad (given it will be clicked) in the next 4 days. Evaluation metric: **MAE**.

rel-f1

- *driver-position* (node regression): predict average finishing position in next 2 months. Evaluation metric: **MAE**.
- *driver-dnf*, *driver-top3* (node classification): predict DNF or top-3 qualification in next 1 month. Evaluation metric: **AUROC**.

rel-trial

- *study-outcome* (node classification): predict if clinical trial achieves primary outcome. Evaluation metric: **AUROC**.
- *study-adverse*, *site-success* (node regression): predict number of adverse patients or site success rate. Evaluation metric: **MAE**.

2.2 Tabular Models

As already mentioned in the introduction, tabular systems refer to the conventional learning approach in which all input data is represented as a single flat table, which is a matrix where each row corresponds to an instance (e.g., a customer, a transaction) and each column corresponds to a scalar feature (e.g., age, total purchases, number of clicks). This format assumes that every instance is independent, identically distributed (**i.i.d.**[2]), and fully described by its feature vector.

Tabular systems are widely used across industry and research due to their simplicity, interpretability, and compatibility with mature, high-performance algorithms such as gradient boosted decision trees (GBDT). Among these systems XGBoost[5] and LightGBM[12] are probably the most adopted ones.

In a tabular pipeline, relational or multi-table databases must first be transformed into a single denormalized table through flattening: a process that involves joining tables (typically via foreign-key relationships) and aggregating related records into fixed-length numerical features. This step requires substantial manual effort and domain expertise, as the quality of the resulting features has a direct impact on model performance.

While powerful and practical, tabular systems cannot natively represent structured or relational dependencies between instances, such as repeated interactions,

[2]The i.i.d. (independent and identically distributed) assumption considers all data points as independent and identically distributed. Standard machine learning tasks (supervised, unsupervised, and semi-supervised) usually assume that each sample is statistically independent from the others, in order to avoid considering dependencies between data points. They also assume that data points are identically distributed to ensure mathematical guarantees of generalization to unseen samples. This assumption does not hold for relational data: as the name suggests, relations are preeminent, and entities are highly interdependent rather than independent.

temporal sequences, or graph-like relationships, and therefore rely entirely on the expressiveness of the engineered features to capture higher-order patterns.

The widespread success of tabular models can be largely attributed to their scalability and the extensive algorithmic optimizations that enable fast and efficient execution[5].

One of the most impactful advancements in gradient boosting methods has been the introduction of XGBoost, a scalable and highly efficient tree-based learning system. Its effectiveness was rapidly recognized in real-world applications and competitive settings: according to Chen and Guestrin (2016)[13], XGBoost was used in 17 out of 29 winning solutions published on the Kaggle blog in 2015, with 8 solutions relying exclusively on it and many others combining it with deep neural networks in ensembles. As mentioned in[5], it was also adopted by all top-10 winning teams in the KDDCup 2015, highlighting its dominant role in tabular data modeling.

2.2.1 Technical Structure of Tabular Models and the Role of Feature Engineering

Tabular models such as XGBoost and LightGBM are based on gradient boosting decision trees (GBDT), an ensemble learning technique that builds a strong predictor by combining multiple weak learners (typically shallow decision trees) through a stage-wise additive process. At each iteration, a new tree is trained to minimize the residual errors (i.e., gradients) of the current ensemble with respect to a differentiable loss function, such as mean squared error for regression or log-loss for classification.

A typical tabular learning pipeline starts from a structured relational database, often composed of multiple interrelated tables (e.g., users, transactions, products). In order to use GBDT models, these tables must be **transformed into a single *denormalized* table**.

The resulting flat table has a matrix structure, where each row corresponds to an instance and each column to a scalar feature. This representation assumes that instances are independent and identically distributed (i.i.d.) and that all relevant information is encoded in the feature vector.

Once the feature matrix is constructed, the GBDT model is trained to predict a target variable. For example, in a fraud detection scenario, each row might correspond to a transaction, with features such as transaction amount, time since last transaction, and customer age. The model learns complex non-linear relationships between features by building a sequence of decision trees, each focusing on reducing the prediction error of the previous ensemble. XGBoost and LightGBM differ in their implementation details: LightGBM uses a histogram-based splitter and leaf-wise tree growth, while XGBoost by default grows trees level-wise (with

an option for leaf-wise).

Despite their high performance, these models have intrinsic limitations: they do not capture dependencies between instances (e.g., temporal or relational links), and rely entirely on the expressiveness of the engineered features. This makes them suboptimal in scenarios where inter-instance interactions are critical, such as social networks, recommendation systems, or time-evolving processes.

2.3 Graph Neural Networks

Graph Neural Network (GNN) refers to any Neural Network working on a graph data. Unlike traditional neural networks that assume a fixed-size input (e.g., vectors, images, sequences), GNNs are capable of learning representations from data where entities (nodes) are connected via arbitrary relationships (edges), such as social networks, molecular structures, knowledge graphs, or interaction networks. The key idea behind GNNs is to iteratively update the representation of each node by aggregating and transforming information from its local neighborhood, thereby capturing both attribute-level information and relational inductive bias. This makes GNNs particularly powerful for tasks such as node classification, link prediction, and graph-level regression or classification.

2.3.1 What is a Graph?

Before delving into machine learning on graphs, it is important to briefly clarify what graphs are and what we mean by graph-structured data.

Due to their ability to model entities and, most importantly, the relationships between them, graphs have become a ubiquitous data structure capable of representing a wide variety of domains and scenarios.

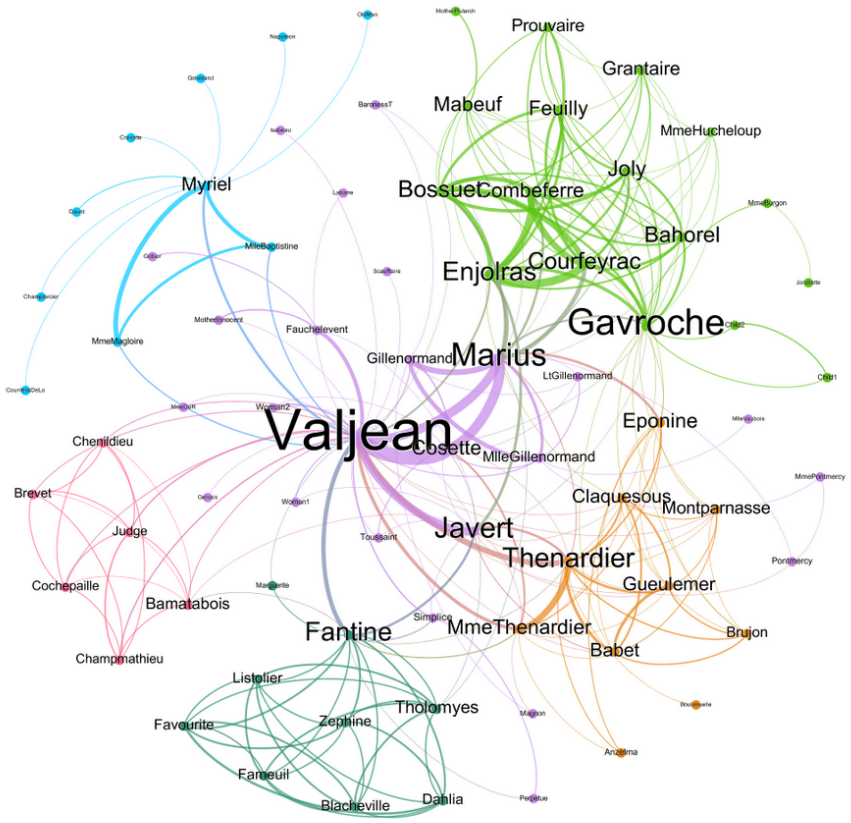


Figure 2.2: Example of a character co-occurrence network inspired by Les Misérables, generated using the Les Misérables dataset (77 characters, 254 weighted undirected edges). Each node represents a character, and edges are weighted by how frequently the characters co-appear. Node size and layout reflect connectivity structure. Although the numerical values are not displayed, the edge thickness visually encodes the weights, with thicker links indicating more frequent coappearances. (Image taken from[16]).

Graphs are more than just a data structure: they offer a mathematical formalism that can be adopted to analyze almost all real-world complex relational systems. Graphs are so powerful and flexible that we **could consider other data types, such as images and texts, as simplifications of graph-data**[3][17]. Therefore, graph structures can be employed in the majority of application areas, including proteomics, image analysis, relational databases (which is the focus of this thesis), scene description, software engineering, natural language processing, and many more, as noted in [10]. The main challenge is to unlock the full potential of this

[3]In fact, images can be viewed as grids of nodes with RGB attributes, while text can be organized into tree- or graph-structured information.

data structure and to improve the learning process over such a mathematical representation, so that we can automatically and simultaneously address tasks in a wide range of applications.

Formally, a graph G=(V, E) is defined by a set of nodes V and a set of edges E between these nodes.

Let $G = (V, E)$ be a graph with $n = |V|$ nodes. Its structure can be described by an adjacency matrix $A \in \mathbb{R}^{n \times n}$, defined as:

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

To represent a graph using an adjacency matrix, we first need to assign a fixed ordering to the nodes so that each row and each column of the matrix corresponds to a specific node in the original graph.

From the definition of the adjacency matrix provided above, we can state the following: undirected graphs produce symmetric adjacency matrices, whereas directed graphs do not.

Graphs can also be weighted, meaning that instead of simply using 0,1 values in the adjacency matrix, we can have arbitrary real-valued entries that express the strength of the connections.

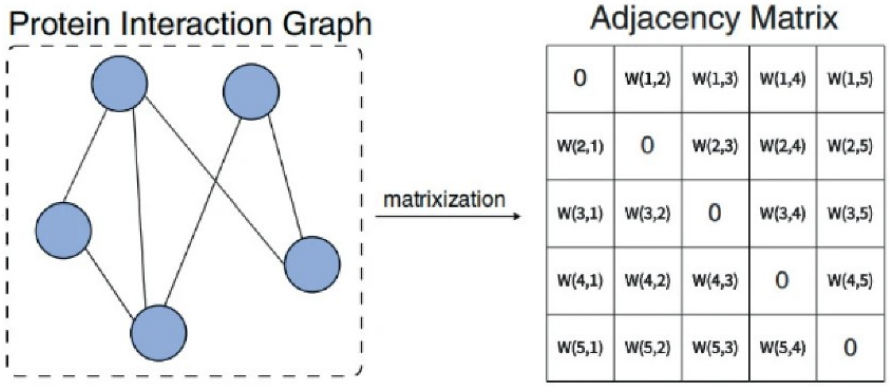


Figure 2.3: Example of weighted graph for a Protein-Protein Interaction Network. A weighted edge in this case may indicate the strength of the association between two proteins[18]. Image taken from[19].

Most of the times adjacency matrix is not able alone to provide all the useful details about a given network, but we usually need to have attribute or feature information associated with the graph. Usually we use a real-valued matrix $\mathcal{X} \in \mathbb{R}^{|\mathcal{V}| \times n}$, where n is the embedding size for a given node[4].

[4]To maintain consistency in the representation, we set the order of X equal to the one of the

In this work, we focus on *heterogeneous graphs*, which contain multiple types of nodes and/or multiple types of edges, each representing different kinds of entities and relationships. Formally, the node set can be expressed as:

$$V = V_1 \cup V_2 \cup \cdots \cup V_T, \quad \text{with} \quad V_i \cap V_j = \emptyset \quad \forall i \neq j,$$

where each V_t corresponds to a distinct node type. Similarly, the edge set is composed of different types of relations:

$$E = \bigcup_{r \in \mathcal{R}} E_r,$$

where $\mathcal{R}$ denotes the set of relation types. Such structures are particularly useful for modeling complex, multi-relational systems such as knowledge graphs, bibliographic databases, or relational information networks. Each relation $r \in \mathcal{R}$ is typically associated with a separate adjacency matrix $\mathbf{A}^{(r)} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{V}|}$, allowing the model to preserve the semantics of individual edge types. Alternatively, the set of adjacency matrices can be represented as a 3D tensor $\mathcal{A} \in \mathbb{R}^{|\mathcal{V}|\times|\mathcal{R}|\times|\mathcal{V}|}$.

Moreover, nodes of different types often come with distinct attribute spaces. For instance, papers and authors in a bibliographic network may have completely different sets of features, requiring type-specific encoding strategies.

2.3.2 Learning on Graphs

Machine Learning tasks are usually classificated as supervised, unsupervised and semi-supervised. When considering graph data types, the situation is not that different, but this distinction is not necessarily the most useful for representing the type of machine learning task on graphs[18]. We, therefore, need to provide a brief overview of the categories of tasks for graph.

- **Node classification:** Node classification is one of the main tasks for graphs. In this setting, the goal is to predict the label y_u for each node $u \in \mathcal{U}$. The label y_u may represent a type, category, or attribute. In this setting, we are only given the true labels for a training set of nodes. Usually, the number of labeled training samples is limited; besides these, the node classification task also considers unlabeled test samples. For such nodes, we do not know the true label, but we are given the relationships they have with other nodes.

  That said, node classification represents a distinctive type of machine learning task. While it resembles standard supervised learning where the goal is to predict labels for individual nodes, only a subset of the nodes in the graph

adjacency matrix.

have ground-truth labels available during training. This partial supervision naturally places node classification within the broader class of *semi-supervised learning* problems.

Although node classification is often described as a semi-supervised learning task (since only a subset of nodes has ground-truth labels during training) it departs from the typical assumptions used in many standard SSL settings. As noted in[18], in classical machine learning theory, semi-supervised methods are frequently analysed under the i.i.d. assumption, where samples are considered independent and identically distributed. This assumption, however, is violated in graphs: nodes are not independent, as their features and labels are correlated through the graph structure. Consequently, node classification can be seen as a semi-supervised problem on non-i.i.d. data, where modeling the relationships between samples is central to the learning process.

Node classification encompasses both **binary** and **multi-class** classification tasks.

A typical example of node classification is predicting the category of users in a social network (e.g., student, researcher, engineer, artist), where each node can belong to one of multiple classes based on its features and connections. In a binary classification setting, one may instead predict whether a user is a spammer or not, based on both their profile information and interaction patterns within the network.

- **Node regression:** Business statistics for reselling companies (such as Amazon) often rely on tasks that require estimating the total amount of budget a user is likely to spend on the platform in the future. Such analyses are inherently regression-oriented and can be addressed through a node regression task over a graph structure that captures all relevant information from the company's dataset.

  Formally, in a node regression task we are given a graph $\mathcal{G} = (\mathcal{V}, E)$, where $\mathcal{V}$ is the set of nodes (or vertices) and $E \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. Each node $u \in \mathcal{V}$ is associated with a feature vector $\mathbf{x}_u \in \mathbb{R}^d$, and only a subset of nodes $\mathcal{V}_{\text{train}} \subset \mathcal{V}$ have known continuous target values $y_u \in \mathbb{R}$. The goal is to learn a function $f : \mathcal{G}, \mathbf{x}_u \mapsto \hat{y}_u \in \mathbb{R}$ that can accurately predict the continuous label y_u for unseen nodes $u \in \mathcal{V}_{\text{test}} = \mathcal{V} \setminus \mathcal{V}_{\text{train}}$.

  As in node classification, the learning process typically leverages both the node features and the graph structure (i.e., the connectivity encoded in E) to make predictions. The objective is usually to minimize a loss function over the training nodes, which is most commonly the Mean Squared Error (MSE):

$$\mathcal{L}_{\text{MSE}} = \frac{1}{|\mathcal{V}_{\text{train}}|} \sum_{u \in \mathcal{V}_{\text{train}}} (y_u - \hat{y}_u)^2 .$$

Node regression is thus the continuous counterpart of node classification, and is particularly useful in scenarios where target values are real-valued quantities, such as prices, ratings, or user spending estimates.

- **Link prediction:** Recommender systems [20, 21] aim to suggest items to customers, and can be formulated as a link prediction task, where the link has the semantics of indicating a user's interest in a specific product.

  Formally, the link prediction task consists in estimating the likelihood of the existence of a link between two nodes $u, v \in \mathcal{V}$ in a given graph $\mathcal{G} = (\mathcal{V}, E)$, where $\mathcal{V}$ is the set of nodes and $E \subseteq \mathcal{V} \times \mathcal{V}$ is the set of observed edges. Given a pair of nodes (u, v), the model must output a score $s(u, v) \in \mathbb{R}$ representing the strength or probability of the existence of an edge between them. The score function $s(\cdot, \cdot)$ is typically parameterized by a neural network that leverages both node features $\mathbf{x}_u, \mathbf{x}_v$ and structural information from the graph.

  The task is commonly trained using a binary classification objective: positive samples are node pairs (u, v) such that $(u, v) \in E$, while negative samples are sampled from $(u, v) \notin E$. The loss function often used is the binary cross-entropy:

$$\mathcal{L}_{\text{BCE}} = - \sum_{(u,v) \in \mathcal{D}} y_{uv} \log \sigma(s(u, v)) + (1 - y_{uv}) \log(1 - \sigma(s(u, v))),$$

  where σ is the sigmoid function and $y_{uv} \in \{0,1\}$ is the ground truth label indicating the presence or absence of the edge. Link prediction is particularly useful in recommendation settings, where the nodes u and v can represent users and items respectively, and the predicted link expresses a potential interaction, such as a purchase or a rating.

  The complexity of this kind of task is highly dependent on the structure of the graph itself. For example, heterogeneous graphs, which may include hundreds of different relation types, are typically more complex to handle.

  One of the most influential approaches to link prediction in knowledge graphs is the TransE model [22], which represents both entities and relations as vectors in a continuous embedding space. The core idea of TransE is to model a relation as a translation between head and tail entities, such that for a true triple (h, r, t), the embedding of the head entity $\mathbf{h}$ translated by the relation vector $\mathbf{r}$ is close to the embedding of the tail entity $\mathbf{t}$; formally, $\mathbf{h} + \mathbf{r} \approx \mathbf{t}$.

This method provides an efficient and scalable way to perform link prediction by learning to score the plausibility of triples via simple vector operations.

- **Graph classification, regression and clustering:** Another important category of graph tasks involves using entire graphs as input and producing a label for each of them. The label can be a real-valued output (in which case we have a *graph regression* task) or a categorical value (a *graph classification* task).

  The aim is to train a model using many independent and labeled graphs, in order to improve its ability to assign the correct label to unseen inputs effectively learning a mapping function from whole graphs to labels. Graph regression and graph classification are perhaps the most direct analogues to standard supervised learning, as each graph is treated as an i.i.d. datapoint associated with a label.

  Graph clustering, on the other hand, is an unsupervised task where the goal is to group nodes into clusters based on their structural or feature similarity, without the need for labeled data.

2.3.3 Core Challenges in Applying Deep Learning to Graph

When it comes to applying Neural Networks to graphs, we can think about simply using the standard Neural Network architectures over these data structures.

To assess whether standard Deep Learning architectures can be applied to graphs, let us assume we are given a graph G, where:

- V is the vertex set
- A is the adjacency matrix (assume binary)
- $\mathbf{X} \in \mathbb{R}^{m \times |V|}$ is a matrix of node features
- v: a node in V
- $N(v)$: the set of neighbors of v.

The easiest way to apply a standard Deep Learning model in this context would be to concatenate the node features with the adjacency matrix, so that each row contains both the features of a node and its adjacency information.

We can then apply a Multi-Layer Perceptron (MLP) to this matrix to initiate the learning process.

Despite being a very simple and straightforward application, this approach presents several issues:

- No flexibility with respect to graph dimensionality: this approach requires a fixed number of input neurons, making it unsuitable for graphs of varying sizes or structures.
- The result depends on the ordering of the matrix: this method is sensitive to the specific order of the nodes in the adjacency matrix. Any permutation of the node ordering can lead to different predictions. As we will see later, this approach does not satisfy the permutation invariance property.

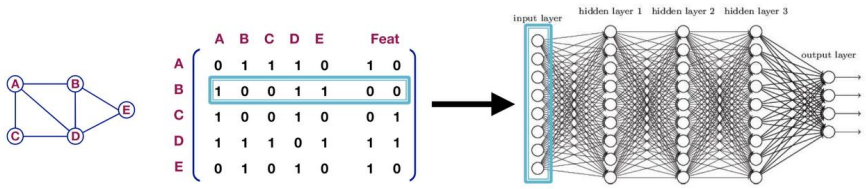


Figure 2.4: Applying an MLP to a graph by concatenating the adjacency matrix A with the node features X (i.e., $[A \,|\, X]$). This naïve approach suffers from two drawbacks: (i) it requires a fixed input size, making it incompatible with graphs of varying order or structure; and (ii) predictions depend on the arbitrary node ordering in A.

In general, using other famous Deep Learning architectures would not change this result: Standard Deep Learning architectures cannot be directly applied to graph. This is primarily due to the **permutation invariance** property of graph data, which renders traditional neural network architectures ineffective without substantial modification.

Permutation invariance means that the output of the learning process should be independent of the order in which we represent the graph[5].

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of elements (e.g., node features), and let $f : \mathcal{X} \to \mathcal{Y}$ be a function that maps a set to an output. We say that f is **permutation invariant** if for any permutation π of the indices $\{1, \ldots, n\}$, the following holds:

$$f(x_1, x_2, \ldots, x_n) = f(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)}).$$

In other words, the output of f remains unchanged regardless of the order in which the inputs are provided. Operations like sum, mean, and max are therefore permutation invariant, since they provide the same output for every node order.

Another key property when dealing with graphs is the **permutation equivariant property**.

[5]For instance, if we model the graph through an adjacency matrix, the output that a certain Machine Learning model provides should be independent of the order of the nodes in the matrix.

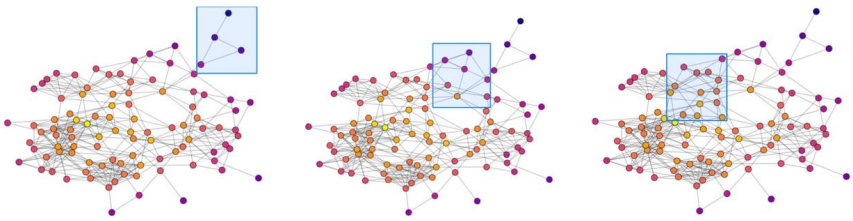


Figure 2.5: Example of applying a fixed-size kernel CNN architecture to a graph. Several challenges emerge:
– Fixed-size window: Choosing the appropriate window size is difficult, as the number of nodes covered may vary depending on the graph's structural properties.
– No fixed locality: Graphs do not have a regular notion of locality or a sliding window mechanism, making it non-trivial to define how the kernel should move across the input.

Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a function, and let $\pi \in S_n$ be a permutation from the symmetric group S_n, which can be represented as a permutation matrix $P_\pi \in \mathbb{R}^{n \times n}$.

The function f is said to be permutation equivariant if for every permutation $\pi \in S_n$ and every input $\mathbf{x} \in \mathbb{R}^n$, it holds that:

$$f(P_\pi \mathbf{x}) = P_\pi f(\mathbf{x})$$

This property guarantees that permuting the input vector and then applying the function yields the same result as applying the function first and then permuting the output.

Example: The elementwise doubling function

$$f(\mathbf{x}) = 2 \cdot \mathbf{x}$$

is permutation equivariant because:

$$f(P_\pi \mathbf{x}) = 2 \cdot (P_\pi \mathbf{x}) = P_\pi (2 \cdot \mathbf{x}) = P_\pi f(\mathbf{x})$$

Figure 2.6: On the left, we see the image of a dog. On the right, the same image is shown, but with its pixels (i.e., nodes) in a random order. Although the pixel values remain the same, changing their order completely alters the result. A CNN architecture relies on this positional bias to perform classification and is therefore not invariant to the order of input nodes.

Are other Neural Network Architectures (e.g. MLP, CNN and RNN) permutation invariant/equivariant?

No, because switching the order of the input leads to different outputs as illustrated in picture2.6. For example, convolutional neural networks (CNNs) are well-defined only over a grid-structured input (such as the images), while reccurent neural networks (RNNs) are well-defined only over sequences.

For this reason, a specific neural network architecture needs to be properly designed to handle graph-structured data. This architecture is called Graph Neural Network, and it consists of multiple permutation equivariant and/or invariant functions.

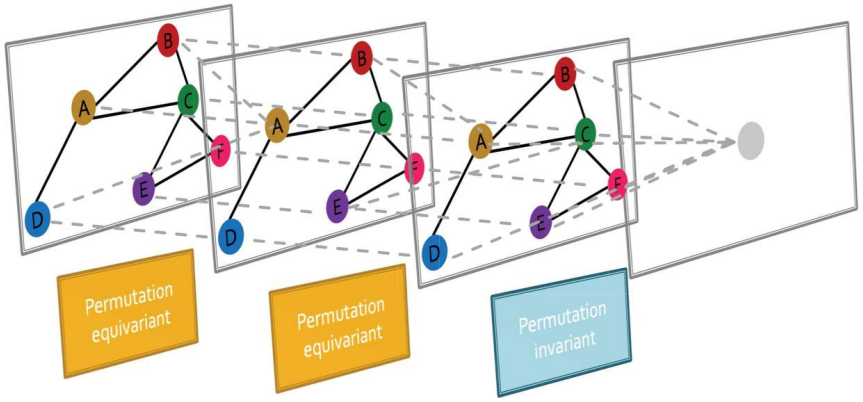


Figure 2.7: Example of a GNN structure. It is composed of several permutation equivariant layers, followed by a final permutation invariant layer that produces a single output. Notice that this is an example of a graph classification task, where a single output is required for the entire graph.

2.3.4 What is a GNN?

A Graph Neural Network (GNN) is a neural framework designed to generate the representation of nodes, also considering the structure of the graph.

The key novelty is that node embeddings are based on local network neighborhoods. In fact, to take into account the graph-structured information, GNNs use a process that can be described in two steps:

1. **Computational graph construction:** During this step, a new graph for a given target node is built to define how the information should be passed through the graph, allowing each node to aggregate information from its neighbors. Each node defines its own computational graph.
2. **Message passing:** prepare and aggregate informations between nodes based on the constructed computational graph.

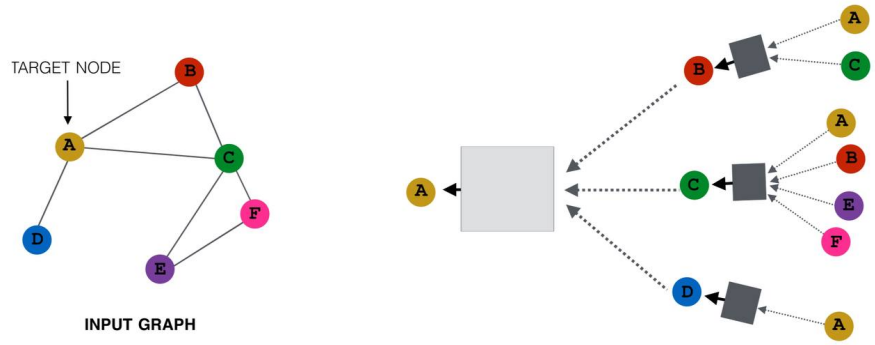


Figure 2.8: Computation graph for a single node (A) induced by the input graph. The embedding of node A is obtained by aggregating information from its neighbors and, recursively, from their neighbors across L message-passing layers (i.e., within its L-hop neighborhood).

The fundamental element that is passed through the graph is called a **message**. A message is the unit of information that a node receives from its neighbors during the learning process. It contains relevant features of the neighboring nodes. The entire GNN model can be seen as a process of **message exchange** and **message aggregation** between neighbor nodes.

Message passing refers to the full computational process that occurs in a Graph Neural Network layer, which typically includes three main steps: *message generation*, where messages are computed based on the features of neighboring nodes, node features and edge attributes; *message aggregation*, where the messages from all neighbors are combined in a permutation-invariant way; and *node update*, where the node's embedding is updated based on the aggregated message and its previous state.

Regardless of the specific GNN architecture, the definition of a GNN is that it uses a form of *neural message passing* in which vector messages are exchanged between nodes, aggregated, and used to update the nodes' embeddings using neural networks [11].

The essential idea of GNNs is to iteratively **update node representations** by combining their neighbors' messages with their own representations from the previous time step.

During each message-passing iteration in a GNN, a hidden embedding $\mathbf{h}_u^{(k)}$ is computed for each node u, considering u's computational graph neighborhood $\mathcal{N}(u)$.

We can provide a general formalization of the GNN learning procedure. We start by initializing the representations for all nodes in the graph with their input features, so that $\mathbf{h}^{(0)} = X$. Then, at each iteration, each node receives messages from its neighbors and aggregates them to update its embedding:

$$a_v^{k+1} = \text{AGGREGATE}^k\left(\left\{\mathbf{h}_u^k : u \in \mathcal{N}(v)\right\}\right) \tag{2.3.1}$$

$$\mathbf{h}_v^{k+1} = \text{UPDATE}^{k+1}\left(\mathbf{h}_v^k, a_v^{k+1}\right) \tag{2.3.2}$$

So, the new embedding for a node v at each iteration is given by:

$$\mathbf{h}_v^{k+1} = \text{UPDATE}^k\left(\mathbf{h}_v^k, \text{AGGREGATE}^k\left(\left\{\mathbf{h}_u^k \mid u \in \mathcal{N}(v)\right\}\right)\right) \tag{2.3.3}$$

Note that `UPDATE` and `AGGREGATE` are **arbitrary** differentiable functions, and $\mathcal{N}(v)$ denotes all the neighbors of node v.

At each iteration k of the GNN, the `AGGREGATE` function takes as input the set of embeddings from the node's neighborhood $\mathcal{N}(v)$ and generates a message $m_{\mathcal{N}(v)}^k$ based on the aggregated information. The `UPDATE` function then combines the message $m_{\mathcal{N}(v)}^k$ with the previous embedding $\mathbf{h}_v^k$ of node v to produce the updated embedding $\mathbf{h}_v^{k+1}$.

After running K iterations of the GNN message passing, we can use the output of the final layer to define the embedding for each node:

$$\mathbf{z}_u = \mathbf{h}_u^K, \quad \forall u \in \mathcal{V}$$

Is crucial to notice that the `AGGREGATE` function must be defined over a set of input messages, rather than a sequence. This requirement makes it permutation invariant, as it guarantees that the aggregation result does not depend on the order in which the input messages are received.

2.4 Node Connectivity in Graphs

An important question to consider is: how far can a message travel through the network? In other words, how many hops away can a node receive information from?

The answer lies in the depth of the GNN, specifically in the number of message-passing layers it employs.

In a GNN with k layers, each node aggregates information from its immediate neighbors at each layer. As a result, **after k iterations of message passing, a node's representation will have incorporated information from all nodes that are within k hops in the graph**. This means that the effective receptive field of a node, which is the portion of the graph it can access, grows with the number of layers.

Therefore, the deeper the GNN (i.e., the higher the value of k), the farther a message can propagate in the graph.

The number of GNN layers should be chosen based on the maximum distance that messages are expected to propagate across the graph.

Similarly to other neural network architectures, one might assume that increasing the number of layers leads to improved performance. However, this is not always true for Graph Neural Networks (GNNs), which suffer from the so-called **over-smoothing problem**[23, 24]: as the number of layers increases, the node embeddings tend to converge to similar or even identical values, thereby losing discriminative power. This phenomenon is rooted in the structural semantics of GNN layers: a GNN with k layers allows each node to aggregate information from nodes at most k hops away. When k approaches the diameter of the graph, the receptive field of each node essentially includes the entire graph. As a result, the receptive fields of different nodes increasingly overlap and, since the embedding of a node is determined by its receptive field, this leads to indistinguishable node representations.

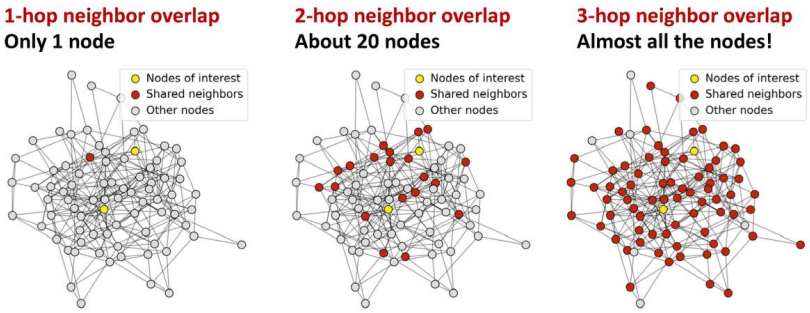


Figure 2.9: This image illustrates how quickly the receptive fields grow as GNN layers are added. This becomes problematic because, once the receptive field covers the entire graph, all nodes tend to aggregate the same information, leading to the well-known over-smoothing problem.

The structural information of the graph is naturally encoded in the message passing mechanism, as messages are exchanged only along the edges present in

the graph. In addition to capturing the graph topology, after k iterations of GNN message passing, each node's embedding also incorporates the features of all nodes within its k-hop neighborhood.

2.5 Popular GNN architectures

So far, we have discussed the general GNN framework. To instantiate it as a concrete architecture, we must specify the AGGREGATE and UPDATE functions.

The first generalized GNN framework was proposed by Scarselli et al. [10] in 2009.

Let $G = (V, E)$ be a graph, where V is the set of nodes and E the set of edges. Each node $v \in V$ is associated with:

- a feature vector $\mathbf{x}_v \in \mathbb{R}^d$,
- a hidden state (embedding) $\mathbf{h}_v \in \mathbb{R}^q$ to be learned.

The model defines a state transition function (applied locally to each node):

$$\mathbf{h}_v = f_{\mathbf{w}}\Big(\mathbf{x}_v,\ \{\mathbf{x}_u\}_{u\in\mathcal{N}(v)},\ \{\mathbf{h}_u\}_{u\in\mathcal{N}(v)}\Big) \tag{2.5.1}$$

where $\mathcal{N}(v)$ denotes the neighbors of v and $f_{\mathbf{w}}$ is a neural function with parameters $\mathbf{w}$. A common non-positional instance is

$$\mathbf{h}_v = \sum_{u\in\mathcal{N}(v)} F(\mathbf{x}_v, \mathbf{x}_u, \mathbf{h}_u) \tag{2.5.2}$$

with F implemented as an MLP. Unlike modern GNNs, this framework updates node states by repeatedly applying the transition until a fixed point is reached.

In contemporary *layer-wise* GNNs, message passing proceeds for a fixed number of layers. A basic formulation is

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right) \tag{2.5.3}$$

where $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$ are trainable matrices and σ is a nonlinearity (e.g., `tanh` or `ReLU`). Thus, at layer k each node updates its embedding using its previous representation and the aggregated representations of its neighbors from layer $k-1$.

The *most basic neighborhood aggregation operation* [18] simply sums a node's neighbor embeddings. However, the sum is highly sensitive to node degrees[6] and can

[6]Meaning that nodes with a high number of neighbors will present a much higher magnitude of $\sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(k-1)}$.

also cause numerical instability and divergence during optimization. To mitigate these issues, one typically averages the contributions of the node's neighborhood. There are several formulations that try to improve equation (2.5.3), but the simplest one is to just take an average rather than the sum:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\frac{1}{|\mathcal{N}(u)|}\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right) \tag{2.5.4}$$

Once node embeddings are computed for each of the nodes, they can be used for downstream tasks.

What we have seen so far is a brief introduction and intuition about how a GNN works mathematically. Several different implementations of GNNs have been proposed; for simplicity, we will primarily focus on the most influential ones in the following paragraphs.

Graph Convolutional Network (GCN)

One of the most commonly used GNN networks is the Graph Convolutional Network. This model employs a slightly different normalization of the neighbors messages defined as:

$$\sum_{v\in\mathcal{N}(u)}\frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}} \tag{2.5.5}$$

This type of normalization is known as **symmetric normalization**, and it is applied to mitigate the effect of node degree imbalance. Each neighbor's embedding $\mathbf{h}_v$ is scaled by the factor $\frac{1}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}}$, where $|\mathcal{N}(u)|$ and $|\mathcal{N}(v)|$ denote the degrees of the target node u and the source node v, respectively. This ensures that information coming from high-degree nodes does not disproportionately dominate the aggregation process.

By applying this symmetric normalization scheme, the message passing function is computed as follows:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)}\sum_{v\in\mathcal{N}(u)\cup\{u\}}\frac{\mathbf{h}_v^{(k-1)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right), \tag{2.5.6}$$

In this equation, each neighbor (including the node itself) contributes to the new representation of node u with a weight inversely proportional to the geometric mean of their degrees. This prevents high-degree nodes from overwhelming the aggregation and stabilizes the optimization.

Note that this formulation does not include an explicit bias term. Although a bias vector $\mathbf{b}^{(k)}$ can be added after the aggregation step to improve expressiveness.

In summary, the GCN model introduced a simple yet powerful framework for learning on graph-structured data through layer-wise propagation and symmetric normalization. Despite its effectiveness in various tasks, the standard GCN has some limitations, such as its inability to handle edge features, its reliance on a fixed aggregation scheme, and its limited receptive field due to shallow architectures. Another important drawback of this model lies in its *transductive setup*: In the standard semi-supervised setting, a GCN is trained with the full normalized adjacency $\hat{A}$ of the *entire* graph. As a result, the representation of a labeled node depends (within K hops) also on unlabeled/test nodes, so the model is conditioned on that specific graph and does not directly handle new nodes/graphs without rebuilding $\hat{A}$ and re-running propagation [7].

These drawbacks have motivated the development of more flexible and expressive models, such as Graph Attention Networks (GATs), GraphSAGE, and other variants that we will explore in the following sections.

GraphSAGE

GraphSAGE (Graph Sample and Aggregate)[25] is a framework for inductive representation learning on large graphs. Unlike transductive methods like GCNs, which require access to the entire graph structure during both training and inference time, GraphSAGE learns a set of aggregator functions that can generalize to unseen nodes or even entirely new graphs. At each layer k, the representation of a node v is computed by aggregating the representations of its neighbors $\mathcal{N}(v)$ from the previous layer $k-1$, together with its own representation:

$$h_v^{(k)} = \sigma\left(W^{(k)} \cdot \text{AGGREGATE}^{(k)}\left(\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\} \cup \{h_v^{(k-1)}\}\right)\right)$$

Here, $h_v^{(k)}$ denotes the embedding of node v at layer k, σ is a non-linear activation function (e.g., ReLU), and $W^{(k)}$ is a learnable weight matrix. The aggregation function can be a simple mean, a pooling operation, or even a recurrent function like LSTM. GraphSAGE enables scalable training by sampling a fixed-size neighborhood for each node instead of considering all neighbors, which is especially useful in graphs with high-degree nodes. This inductive approach makes GraphSAGE particularly well-suited for dynamic or evolving graphs, and for tasks where generalization to unseen data is crucial.

[7] **Transductive** learning uses the test inputs (here, the structure and features of test nodes) at training time, though not their labels. **Inductive** learning instead learns a local aggregation function that can be applied to unseen nodes/graphs without access to them during training (e.g., GraphSAGE-style sampling).

Graph Attention Network (GAT)

One of the most promising approches has been to try to adapt the self-attention mechanism[26] in order to improve the aggregation phase in GNN.

The first GNN architecture to adopt the self-attention mechanism was Veličković et al.[27] in 2018. They defined the Graph Attention Network (GAT), which uses a self-attention mechanism[8] to compute attention scores for each neighbor of a node. These weights indicate the importance to be given to each of the neighbours of a node. In the previous formulation, the weights assigned to the neighbors are implicit, yet still present. In Equation (2.5.6), in fact, the weights are equal for all nodes and are given by $\frac{1}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}$. Consequently, the attention scores in this case are determined solely by the structural properties of the graph, by the node degrees. Self-attention mechanism improves this mechanism allowing the network to decide which are the neighbours that are more influentials for obtaining the output.

To provide a simpler interpretation of the attention mechanism adopted in the GAT architecture we can rewrite formulation(2.5.6) as:

$$\mathbf{h}_u^{(k)} = \sigma\left(\sum_{v \in \mathcal{N}(u) \cup \{u\}} \alpha_{uv} \mathbf{W}^{(k)} \mathbf{h}_v^{(k-1)}\right), \tag{2.5.7}$$

Where:

$$\alpha_{uv} = \frac{1}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \tag{2.5.8}$$

This corresponds to the weighting factor (or importance) of node v's message to node u and is defined explicitly based on the node degree property. It is important to notice that all neighbors of node v are equally important in the GCN architecture.

Attention is inspired by cognitive attention and is based on the idea that a Neural Network should devote more computing power on the most significative and discriminative part of data[26]. What is the most important part of data should be learned during the training process.

The main improvement introduced by the attention mechanism in GNN message passing lies in the idea that the attention assigned to each neighbor should be learned during training and it should take into account both the source and destination node features, rather than being constant.

[8]Note that this mechanism is not canonical Transformer self attention[26]: GAT uses an additive, edge local scoring restricted to 1 hop neighbors with a single linear projection and a learned attention vector, rather than separate Q, K, and V with a global scaled dot product. This simplification reduces computation to $O(|E|)$ and aligns the operation with the sparsity of graph structure.

To compute these coefficients, we associate each edge (u, v) with a learnable attention score e_{uv}, which expresses how relevant node v is to node u. At first, we assume that these scores e_{uv} are already available. To ensure that the attention weights are comparable across different neighborhoods, we normalize them using the softmax function obtaining the attention scores α_{uv}:

$$\alpha_{uv} = \frac{\exp(e_{uv})}{\sum_{k \in \mathcal{N}(u)} \exp(e_{uk})} \tag{2.5.9}$$

These normalized **attention coefficients** α_{uv}, are then used to compute a weighted aggregation over the neighborhood of u.

But how do we actually compute the attention scores e_{uv}?

There are multiple ways to define the function that computes e_{uv} from the input features of nodes u and v.

A straightforward strategy consists in concatenating the linearly transformed embeddings of the source and destination nodes, and then passing the result through a single-layer neural network:

$$e_{uv} = a(\mathbf{W}^l \mathbf{h}_u^{l-1}, \mathbf{W}^l \mathbf{h}_v^{l-1}) = \text{Linear}(\text{Concat}(\mathbf{W}^l \mathbf{h}_u^{l-1}, \mathbf{W}^l \mathbf{h}_v^{l-1})) \tag{2.5.10}$$

where $\mathbf{W}^l$ is a learnable weight matrix shared across all nodes.

The parameters of the function a are trained jointly in an end-to-end fashion.

This approach is agnostic to the specific choice of the attention mechanism a, meaning that we can implement a in various ways. The model remains valid as long as:

1. The function a returns a single real-valued score;

2. The attention scores can be normalized using the softmax function.

In the original paper[27], the authors adopted an approach in which they first concatenated the transformed node embeddings and then applied a linear projection followed by a non-linear activation (LeakyReLU).

Since attention scores are sometimes difficult to make converge, we can further define and use a **multi-head attention** mechanism, which helps stabilize the learning process of the attention itself. This consists in applying multiple independent attention mechanisms (or "heads"), each with its own set of learnable parameters. Each head computes separate attention coefficients and transforms the neighbor embeddings accordingly.

Let K be the number of attention heads. For each head $i \in \{1, \ldots, K\}$, a different attention coefficient $\alpha_{uv}^{(i)}$ and weight matrix $\mathbf{W}^{(i)}$ are used. The output of

each head is computed independently as:

$$\mathbf{h}_v^{(l)[i]} = \sigma\left(\sum_{u \in \mathcal{N}(v)} \alpha_{vu}^{(i)} \mathbf{W}^{(i)} \mathbf{h}_u^{(l-1)}\right) \tag{2.5.11}$$

The final output of the layer is obtained by aggregating the outputs of all heads:

$$\mathbf{h}_v^{(l)} = \text{AGG}\left(\mathbf{h}_v^{(l)[1]}, \mathbf{h}_v^{(l)[2]}, \ldots, \mathbf{h}_v^{(l)[K]}\right) \tag{2.5.12}$$

where AGG denotes an aggregation function, which can be either **concatenation**, **summation**, or others, depending on the specific implementation or position within the architecture (e.g., summation is commonly used in the final layer).

This design improves the model's expressiveness and robustness by capturing information from multiple perspectives.

The GAT model presents several key advantages with respect to the previously presented ones:

- **Differentiated neighbor weights:** the neighbour weights depend on their *importance* for the downstream task.
- **Time- and space-efficient:** A key difference between a GAT and how self-attention layers work in other neural architectures, such as Transformers[26], is that in the latter, the attention score is computed for every pair of nodes, not just adjacent ones[9]. This computation is much more expressive, as it allows the model to immediately capture the importance of any node relative to another, even if they are far apart in the graph structure. However, it is also extremely time-consuming and requires a large amount of memory to store all pairwise computations. In contrast, GATs compute attention scores only between a node and its immediate neighbors at every iteration, effectively ignoring the rest of the graph. This leads to significantly more efficient computations in both time and space.
- **Permutation-invariant:** as previously discussed, permutation invariance is a key property that any aggregation operation over a graph should satisfy. Whenever an operation aggregates node-level information into a single representation, this property should hold. The attention mechanism used in GATs preserves this property.
- **Parallelization:** Multi-head attention improves stability without requiring significantly more execution time. This operation can, in fact, be fully parallelized.

[9]By adjacent nodes we mean nodes that are directly connected through (at least) one edge.

GAT models with multihead self attention are very closely related to Transformer architectures[26]. The Transformer architecture is one of the most important across many fields and has been a key driver of large state of the art NLP systems such as BERT[28]. In fact, if we consider a fully connected graph[10] a GAT layer becomes *functionally equivalent* to a Transformer encoder *provided that* the attention scoring uses the scaled dot product with distinct query, key, and value projections (rather than the additive score with a single projection and an attention vector), the value projection is not tied to the key projection, and the usual residual, normalization, and positionwise feed forward blocks are included. Under these conditions both models rely on multihead self attention to compute contextualized representations by aggregating information from all other elements (nodes in GAT, tokens in Transformers). The key components in both cases are the computation of attention coefficients, softmax based normalization, and aggregation via weighted sums of linearly projected inputs. The remaining structural difference is connectivity: Transformers assume full pairwise interaction, whereas standard GAT restricts attention to a node's 1 hop neighborhood given by the graph topology. In this sense a Transformer encoder can be viewed as graph attention on the complete graph, and a GAT layer as its sparse, topology restricted counterpart that generalizes the computation to arbitrary graphs.

This architecture has significantly advanced the state of the art in Graph Neural Networks, not only because of its expressive power, but also, and perhaps more importantly, due to the radical shift in perspective: leveraging an architecture that already performs well in another domain (such as the Transformer in Natural Language Processing) and adapting it to work effectively with graph-structured data.

Attention mechanism allows model to focus on the most salient part of data. It has been demonstrated the effectiveness in a variety of applications, such as text analysis[29], knoledge graph[30] and image processing[31].

Building on this intuition, several models have since explored how to integrate the self-attention mechanism directly within the Graph Neural Network paradigm. One notable line of research involves the development of Transformer-based models for graphs, such as **Graphormer** [32]. Unlike traditional GNNs like GAT [27], which restrict attention to a node's immediate neighbors, Graphormer removes this constraint and enables *global attention* across all nodes in the graph. To compensate for the lack of topological locality, structural biases are introduced through encoding schemes that inject distance (e.g., shortest-path), centrality, and edge-type information directly into the attention computation.

Graphormer achieves state-of-the-art performance on large-scale benchmarks by

[10]A fully connected graph is a graph where each node is connected to every other node.

leveraging these structural encodings to inform the attention weights, effectively allowing each node to attend to all others in a way that remains sensitive to the graph's geometry. This general strategy has inspired a variety of other architectures[11] which differ in the specific structural biases they encode or the ways they limit the attention scope to preserve sparsity and computational efficiency. These approaches collectively mark a shift toward bridging the gap between message-passing GNNs and fully attention-based models, aiming to combine the strengths of both paradigms.

That said, the original GAT model was designed to operate only on homogeneous graphs, where all nodes share the same type and all edges represent the same kind of relation. While this simplifies the design and computation of the attention mechanism, it also imposes strong limitations in scenarios where data naturally exhibit multiple types of entities and relations. In many real-world applications, such as knowledge graphs, recommendation systems, and multi-relational networks, the graph is inherently heterogeneous: nodes may represent different entity types (e.g., users, items, events) and edges may encode various semantic relations (e.g., purchase, review, attend).

In the following section, we will discuss an extension of the GAT model for heterogeneous graphs, capable of explicitly modelling and leveraging the diversity of node types and relation types present in the data.

Heterogeneous Graph Attention Network (HAN)

The heterogeneity introduces rich semantic concepts and poses significant challenges for designing a graph neural network.

HAN[33] tries to extend the attention mechanism defined in GAT into heterogeneous graph using a **hierarchical attention** mechanism. Differently from GAT, instead of only considering the **node-level attention**, which aims to learn the importance of node's neighborhood, HIN also adopt a **semantic-level attention** mechanism.

The idea can be expressed as follows:

- We begin with a set of **meta-paths**, which are **predefined** before applying the model. A **meta-path** Φ in a heterogeneous information network (HIN) is defined as a path composed of a sequence of node types and edge types in the form:
$$\Phi : A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \dots \xrightarrow{R_l} A_{l+1}$$
where $A_i \in \mathcal{A}$ are node types and $R_i \in \mathcal{R}$ are relation types. A meta-path defines a composite semantic relation between nodes of type A_1 and A_{l+1}

[11] Sometimes referred to as *Graph Transformers*.

by chaining together atomic relations $R_1, R_2, \ldots, R_l$. It is also required that $A_1 = A_{l+1} = A_{\text{target}}$, i.e., the meta-path starts and ends on the target node type. For instance, the meta-path APA (Author–Paper–Author) defines a relation between two authors who have co-authored a paper.

- For each meta-path Φ, a corresponding **meta-path-based homogeneous graph** G_Φ is constructed, where all nodes are of the same target type. An edge exists between two nodes in G_Φ if there is a path between them in the original heterogeneous graph that follows the meta-path Φ. This induced homogeneous graph allows attention-based message passing to operate within a well-defined semantic context.

- Before performing meta-path-based aggregation, it is necessary to *align the feature spaces* of the different node types present in the original heterogeneous graph. This is not done to make the graph structurally homogeneous, but rather to ensure that node representations from different types are comparable and can be processed with shared parameters in subsequent attention and aggregation steps.

  The authors achieve this alignment through a **node-type-specific transformation**:

  $$\mathbf{h}'_i = \mathbf{W}_{\phi(i)}\mathbf{h}_i \tag{2.5.13}$$

  where:

  - $\mathbf{h}_i \in \mathbb{R}^{F_{\phi(i)}}$ is the original input feature vector of node i,
  - $\mathbf{W}_{\phi(i)} \in \mathbb{R}^{d \times F_{\phi(i)}}$ is a learnable transformation matrix specific to the node type $\phi(i)$,
  - $\mathbf{h}'_i \in \mathbb{R}^d$ is the transformed feature vector in the shared latent space of dimension d.

  By applying node-type transformations each node type share a common feature space that allow them to be comparable.

- **Node-level attention**: for each meta-path, we consider its corresponding homogeneous graph and apply the standard GAT model to compute attention over neighboring nodes. For a meta-path Φ, the attention coefficients are computed as:

  $$e_{ij}^{\Phi} = att_{node}(\mathbf{h}'_i, \mathbf{h}'_j; \Phi) \tag{2.5.14}$$

  Where att_{node} denotes the deep neural network which performs the node-level attention. Given meta-path Φ, att_{node} is shared for all meta-path based node pairs.

These scores are then normalized using the softmax function, as in the original GAT formulation [27]:

$$\alpha_{ij}^{\Phi} = \frac{\exp(e_{ij}^{\Phi})}{\sum_{k \in \mathcal{N}_i^{\Phi}} \exp(e_{ik}^{\Phi})} \tag{2.5.15}$$

The final node representation for meta-path Φ is obtained by aggregating information from neighbors:

$$\mathbf{z}_i^{\Phi} = \sigma \left(\sum_{j \in \mathcal{N}_i^{\Phi}} \alpha_{ij}^{\Phi} \cdot \mathbf{h}'_j \right) \tag{2.5.16}$$

Here $\mathbf{z}_i^{\Phi}$ indicates the learned embedding of node i for the meta-path Φ.

In the original paper, the authors further extended this mechanism to **multi-head attention**, by repeating the node-level attention computation K times and then concatenating the results, as follows:

$$\mathbf{z}_i^{\Phi} = \|_{k=1}^{K} \sigma \left(\sum_{j \in \mathcal{N}_i^{\Phi}} \alpha_{ij}^{\Phi} \cdot \mathbf{h}'_j \right) \tag{2.5.17}$$

- **Semantic-level attention**: after applying node-level attention for each meta-path $\Phi \in \mathcal{P}$, every node i is associated with a set of meta-path-specific embeddings $\{\mathbf{z}_i^{\Phi}\}_{\Phi \in \mathcal{P}}$. Since different meta-paths capture different semantic relations in the heterogeneous graph, not all of them are equally informative for the downstream task. The goal of semantic-level attention is to learn a global importance weight for each meta-path and aggregate the corresponding embeddings accordingly.

  First, for each meta-path Φ, we compute an importance score w_{Φ} as:

  $$w_{\Phi} = \frac{1}{|V|} \sum_{i \in V} \mathbf{q}^{\top} \cdot \tanh\left(\mathbf{W} \cdot \mathbf{z}_i^{\Phi} + \mathbf{b} \right) \tag{2.5.18}$$

  where:

  – $\mathbf{W} \in \mathbb{R}^{d' \times d}$ and $\mathbf{b} \in \mathbb{R}^{d'}$ are learnable parameters of the semantic attention layer,

  – $\mathbf{q} \in \mathbb{R}^{d'}$ is a learnable query vector shared across all meta-paths,

  These scores are then normalized across all meta-paths using a softmax:

  $$\beta_{\Phi} = \frac{\exp(w_{\Phi})}{\sum_{\Phi' \in \mathcal{P}} \exp(w_{\Phi'})} \tag{2.5.19}$$

where β^{Φ} represents the normalized semantic-level attention weight for meta-path Φ.

Finally, the semantic-level aggregation produces the unified node representation by weighting and summing the meta-path-specific embeddings:

$$\mathbf{Z} = \sum_{\Phi \in \mathcal{P}} \beta_{\Phi} \cdot \mathbf{z}_i^{\Phi} \tag{2.5.20}$$

This mechanism allows the model to automatically identify which meta-paths are most relevant for the task, effectively combining multiple semantic contexts into a single, discriminative embedding. For example, in a bibliographic network, a metapath capturing *co-authorship* relations (Author–Paper–Author) might be more predictive for collaboration analysis than one capturing *shared publication venues* (Author–Paper–Venue–Paper–Author), and the semantic attention mechanism will learn to assign higher weight to the former.

This architecture not only enables the modeling of heterogeneous graphs[12], but also enhances interpretability. By leveraging semantic-level attention, it becomes possible to identify which meta-paths contributed most significantly to the final prediction (i.e., those with the highest attention scores), and within those paths, which neighboring nodes had the greatest influence.

Despite its strengths, the HAN model presents a significant limitation that must be acknowledged: the meta-paths are assumed to be predefined and **fixed**. The model does not provide any guidance or mechanism for automatically discovering or selecting the most appropriate meta-paths for a given task or dataset. This is a critical issue, as the quality and robustness of the learned node representations are highly dependent on the choice of meta-paths. Poorly chosen meta-paths may lead to suboptimal performance or even degrade the effectiveness of the attention mechanism. As such, the success of the model relies heavily on expert knowledge or prior domain insights to define meaningful and semantically rich meta-paths. This assumption introduces a non-trivial manual component into an otherwise fully learnable architecture, limiting its applicability in settings where such domain knowledge is unavailable or difficult to formalize.

2.6 Explainable GNNs

Graph Neural Networks (GNNs) have become the de facto standard for many predictive tasks involving relational data. They have been increasingly applied to a wide range of domains where data can be represented as a network, such as social

[12]Please note that real-world graphs typically involve multiple types of nodes and edges.

networks [34, 35], recommender systems [36], physical systems [37], knowledge graphs [38], drug discovery [39], and many others.

Despite their rapid development, success, and widespread adoption, GNNs often exhibit a black-box nature in their predictions. The lack of interpretability in GNN models poses a critical challenge in domains where transparency, accountability, and trustworthiness are essential. While GNNs are capable of learning complex relational patterns and achieving state-of-the-art performance, their internal reasoning process is often opaque to human observers. This opacity hinders the ability of stakeholders to understand why a given decision has been made, making it difficult to detect biases, validate correctness, or ensure compliance with regulatory requirements.

In safety-critical domains, such as healthcare or drug discovery, this issue can have significant consequences. Consider, for example, the use of GNNs in drug discovery for predicting potential drug-target interactions [39]. A GNN might predict that a specific compound has a high likelihood of binding to a protein associated with a given disease. However, without an interpretable explanation of which molecular substructures or interaction pathways led to this conclusion, medicinal chemists are left without the necessary insights to assess the plausibility of the prediction. This not only limits the utility of the model for guiding experimental validation, but also increases the risk of pursuing costly and time-consuming laboratory experiments based on spurious correlations learned from the training data.

Similarly, in recommender systems [36] deployed in social networks, the lack of interpretability can result in user distrust and reputational damage. For instance, if a GNN-based system recommends politically sensitive or potentially harmful content to a user, platform operators may be unable to provide a satisfactory justification for why that recommendation was made, heightening the risk of perceived discrimination and bias, which can lead to reputational harm for the company.. In regulated sectors, such as financial services, this opacity could directly conflict with explainability requirements imposed by legislation (e.g., the European Union's AI Act or the General Data Protection Regulation's "right to explanation").

Therefore, despite their predictive power, the "black-box" nature of GNNs presents a significant barrier to their adoption in domains where human oversight, compliance, and informed decision-making are non-negotiable. Addressing this limitation requires the development of interpretable GNN architectures that can provide faithful, human-understandable explanations of the model's predictions even if this means sacrificing performance.

In this context, **interpretability** refers to the ability to understand and trace how the model produces its results, both at the individual and at the global level.

Interpretability can be categorized into two main families, depending on the granularity of the explanation:

- **Local interpretability**, which concerns understanding the prediction for a

specific node. In the context of GNNs, this can be achieved by identifying which specific relations and neighboring nodes contributed most to a certain output, along with their associated attention weights and structural statistics. Such analysis makes it possible to trace exactly *which edges and intermediate nodes* influenced the prediction and *to what extent*. In particular, given a certain node embedding, one can trace back and identify the most influential relational connections that the model exploited during the message-passing phase, thereby reconstructing the "relational logic" that the network considered most relevant for building that embedding. Furthermore, applying self-attention mechanisms enables the assignment of an importance score to each node within the aggregation process. This allows not only for a precise explanation of the most influential relations (edges) for a given output, but also for a quantification of the contribution of each individual node to that output.

- **Global interpretability**, which aims to understand the model's behavior across the entire dataset. Unlike local interpretability, the goal here is not to explain why a specific node received a particular embedding, but rather to identify which types of relations and structural patterns are generally the most influential for the model. This can be achieved by tracking the frequency of usage and overall impact of different relation types or structural motifs during training. Such analysis provides a high-level view of the relational structures that the model relies on, allowing researchers and practitioners to extract domain-relevant knowledge and validate modeling assumptions. This capability is particularly valuable for extracting dataset-level statistics, as it enables practitioners to determine which tables, features, and instances are most relevant in a general sense.

Regardless of the chosen granularity, explanations are generally classified into two broad categories: (1) Factual and (2) Counterfactual. As noted in [40], factual methods aim to find an explanation in the form of input features that exert the greatest influence on the prediction. In practice, they describe why the model made a certain prediction. Counterfactual methods, on the other hand, describe how the input graph should change in order to alter the prediction. In the following, we will focus on factual explanations, as our main objective is to explain the reasoning of the model after a prediction.

Both factual and counterfactual approaches can be further classified into: (i) post-hoc explainability methods, which aim to interpret model predictions after training, most often at the instance level, by analyzing the factors that led to a specific prediction; and (ii) self-explainable models, which are designed to provide explanations as part of their prediction process.

Post-hoc Interpretation Post-hoc interpretation refers to a family of techniques applied *after* a model has been trained, with the goal of explaining its predictions without altering the underlying architecture. These methods can also be categorized as **white-box** or **black-box**, depending on whether they require access to the model's internal parameters. In white-box settings, the explainer needs information about the model's internal parameters and architecture, while in black-box settings, explanations are derived solely from the model's inputs and outputs without such access.

Typical examples of post-hoc interpretability include *GNNExplainer*[41], *PG-Explainer*[42], and gradient-based saliency methods, all of which aim to highlight important subgraphs, features, or node neighborhoods that significantly influenced the output. All these methods fall under the white-box category, since they require access to the model's internal representations, such as gradients. In particular, decomposition-based methods explain a prediction by decomposing it into contributions from individual input components (e.g., nodes, edges, features), whereas gradient-based methods compute the sensitivity of the output with respect to the inputs by backpropagating gradients through the model.

An example of a black-box approach is given by surrogate methods, which do not require access to the model's internal parameters. Instead, they approximate the behavior of the target model by training an interpretable surrogate model (e.g., a decision tree or a simpler GNN) on the same input–output pairs. The explanations are then derived from the surrogate model, under the assumption that it faithfully mimics the original model's decision boundaries.

In practice, post-hoc GNN explainers often operate by *masking* parts of the input graph and measuring the impact on the model's output. For instance, GNNExplainer learns continuous masks over edges and features within the computational subgraph relevant to the target node or prediction. These masks are optimized to retain only the smallest set of edges and features that preserve the model's original output with high fidelity. The resulting high-weight edges and features form a human-interpretable explanation, pointing to the specific relationships and node attributes that most strongly influenced the decision.

Post-hoc techniques offer the advantage of being *model-agnostic*: they can be applied to any existing trained model without the need for re-design or re-training. However, their explanations are not intrinsically tied to the model's decision-making process; instead, they approximate it based on the observed behaviour. Therefore, the explanation may not faithfully represent the true reasoning process of the model. Post-hoc methods often identify correlations rather than causal relationships. Small changes in the input can lead to significantly different explanations, undermining their reliability. Furthermore, since these architectures are applied after the actual training, they require additional optimization loops, which can be computationally expensive when dealing with large graphs or when explanations are needed for

many instances.

Self-explainable models A model is considered *self-explainable* (also said "ante-hoc") when its decision making process is inherently transparent, such that the reasoning behind each prediction can be directly derived from the model's internal computations, without the need for external, post-hoc interpretability techniques. In other words, the model is designed in such a way that the same mechanisms used to produce the output also generate a faithful and human-understandable explanation of that output.

In traditional machine learning and deep learning settings, most models are *not* self-explainable. Their internal representations and intermediate computations are opaque, and understanding why a specific prediction was made typically requires the use of external (post-hoc) methods, but these methods usually produce an explanation that is *plausible* but not necessarily faithful to the exact reasoning of the model. This approach suffers from the well-known *faithfulness problem*: the explanation might not perfectly align with the true internal logic used during prediction. In contrast, a self-explainable model **integrates interpretability *by design***. Its architecture and computation steps are explicitly structured so that intermediate outputs are semantically meaningful and directly linked to the final decision. This means that the path from input to prediction is not a black box, but rather a sequence of transparent, interpretable transformations. In the case of graph neural networks, self-explainability often involves mechanisms that explicitly identify which structural patterns, node features, or relational connections contributed most to the prediction.

Self-interpretable methods are usually composed of two modules: a subgraph extraction module, which identifies an informative subgraph G_s from the input graph G, and a prediction module, which uses G_s to produce the final prediction. The advantage of this approach is that G_s serves both as the basis for prediction and as a direct source of interpretability. Both modules are trained jointly, avoiding the need for additional computation to explain a given prediction. However, this improved interpretability and computational efficiency often comes at the cost of predictive performance.

2.6.1 Meta-path based models

There are many ways to inject interpretability into GNNs; inspired by HIN [33], we now focus on meta-path based GNN architectures.

Meta-paths are not only designed to improve the performance of the model, but also to build interpretable models. They can, in fact, enhance model performance by guiding the message-passing process along semantically meaningful relational patterns. In practice, this means constraining the flow of information in the GNN so

that each node aggregates messages only from neighbors connected through specific sequences of relations, rather than from the entire heterogeneous neighborhood. By restricting the message-passing operation to these well-defined relational subspaces, the model focuses on information that is most relevant to the task, thereby reducing the influence of noisy or irrelevant edges, capturing long-range dependencies more effectively, and incorporating domain-specific knowledge directly into the structural representation of the graph. This not only facilitates more stable and data-efficient learning, but also enables the GNN to better exploit high-level semantic relationships that may be difficult to capture through generic graph convolutions alone.

By acting as inductive biases that guide the learning process, meta-paths, implicitly serve as interpretable and semantically meaningful explanations of the model's relational reasoning. Each meta-path, infact, explicitly encodes a chain of typed relations, it enables a form of structured interpretability that highlights which types of interactions are most relevant to the model's predictions. This includes both local interpretability by identifying which meta-paths and intermediate nodes contributed to a specific prediction and global interpretability by analyzing which relational patterns are generally most informative across the dataset.

By analyzing the selected meta-paths, the associated attention weights $\alpha_v^{(k)}$, and the contribution of intermediate nodes along the path, we can trace precisely which relational evidence led to a specific prediction. This level of interpretability is particularly valuable in high-stakes or data-sensitive domains, where model transparency is crucial.

For example, in a bibliographic network, a meta-path such as Author–Paper–Venue–Paper–Author might strongly contribute to predicting collaboration likelihood between two researchers. By inspecting the attention weights and the intermediate nodes (specific papers and venues), it becomes possible to explain the prediction in human-understandable terms: *"The model predicts a high collaboration probability because both authors have published in the same venue through related papers"*.

By leveraging meta-paths in this way, we move from opaque reasoning to fully transparent, white-box relational inference.

The results obtained when using meta-paths for heterogeneous graphs strongly depend on the informativeness of the chosen meta-paths. In order to build a powerful GNN model based on meta-paths, we should design meta-paths that, on their own, are able to produce rich node embeddings that can readily support accurate predictions. The performance therefore heavily depends on the quality of these paths, as message passing in the GNN will operate exclusively along them. An important point of discussion concerns how to select informative meta-paths.

Existing approaches typically rely on domain experts to manually define relevant meta-paths in advance [33, 43, 44, 45].

Since the main aim of our approach is to avoid manual feature engineering and

to implement an automatic and effortless solution applicable to any real-world scenario and any relational task, a sort of "film perfectly adhering to the relational context, ready to be applied in any setting", we cannot rely on domain knowledge and manual meta-path design, as this would completely invalidate our objective.

Therefore, our method should be designed to **operate directly on raw relational data**. This eliminates the need for manual preprocessing and enables seamless adaptation across diverse domains. In this setting, **flexibility** is not just desirable, it **is essential**. A truly general-purpose model must be capable of reasoning over arbitrary relational structures without requiring expert intervention or task-specific tuning.

To date, only a limited number of studies[50, 46, 49, 48, 47] have explored the problem of automatic meta-path selection.

MPS-GNN

Among the few attempts to automate meta path discovery, we were fascinated by the work of Ferrini et al. [50], MPS GNN (Meta Path Selection Graph Neural Network), which stands out as a fast, effective and fully-automated approach. Instead of relying on manually defined meta-paths, MPS-GNN introduces a task-driven mechanism for discovering informative meta-paths and encoding them into a self-explainable message-passing architecture.

In *MPS-GNN*, self-explainability is achieved through the meta-path selection and attention mechanisms. During the forward pass, the model selects a subset of meta-paths that it deems most relevant for a given task. Each meta-path defines a structured message passing scheme, along which information flows from neighbour nodes to the target node. The importance of each meta-path is quantified through learned attention weights, which reflect the model's assessment of its predictive value. Since these meta-path choices and attention scores are produced as part of the normal prediction process, they inherently explain the model's reasoning. There is no need to introduce any auxiliary interpretability module or post-hoc analysis: the explanation is built into the model's operation and is guaranteed to be faithful. As a result, the model does not only produce an embedding or a prediction, but also a structured and verifiable account of *why* that embedding or prediction was obtained.

Each meta-path $\mathcal{P}$ induces a multi-hop semantic adjacency matrix $A^{(\mathcal{P})}$ defined such that:

$$A^{(\mathcal{P})}_{uv} = 1 \quad \text{if there exists a path from } u \text{ to } v \text{ following } \mathcal{P}, \text{ else } 0$$

MPS-GNN performs a selection of the most useful meta-paths for the downstream task, without requiring the model to be trained on every possible combination of relations, which would be too costly in terms of computational efficiency. Let $\mathcal{C}$ be

the set of all meta-paths up to length L. The model selects a subset $\mathcal{M} \subset \mathcal{C}$ of K meta-paths that yield the highest prediction performance.

To evaluate the informativeness of each candidate relation at each step of the greedy meta-path construction, MPS-GNN adopts a **weighted multi-instance classification** approach based on the concept of **bags of nodes**.

In practice, the algorithm starts by initializing a *bag vector* whose elements correspond to the features of the instances in the target table[13]. At the same time, a *label vector* of the same length is created, where each entry stores the ground-truth label y_v of the corresponding target node v. Importantly, this label vector remains *fixed* across all iterations of the algorithm, since the prediction target for each instance does not change as the meta-path is extended.

At iteration $t = 0$, each element of the bag vector contains a single node: the corresponding target node v from the target table. The algorithm then considers all possible relations (edge types) that originate from the current node type present in the bags. For every candidate relation $r \in \mathcal{R}$, it creates a new temporary meta-path by appending r to the current prefix $\mathcal{P}_t$, producing an extended path:

$$\mathcal{P}_{t+1} = \mathcal{P}_t + r.$$

For each candidate relation r, the extension process is applied to all target nodes in parallel. Starting from a target node v, the algorithm follows the relation r to retrieve all nodes that are directly reachable from v along that relation type. These reachable nodes form a *bag* B_v associated with v. From a data structure perspective, in the bag vector, the position previously occupied by the single node v is now replaced with the set of nodes $\{u_1, u_2, \ldots, u_k\}$ reached via the one-step traversal defined by r.

Repeating this for every target node in the training set yields a complete bag vector for the extended path $\mathcal{P}_{t+1}$. Each bag B_v inherits the label y_v from its original target node v in the fixed label vector, ensuring that all instances within the same bag are associated with the same prediction objective. This process is repeated iteratively, with each new iteration considering further one-step extensions of the meta-path, while the label vector continues to provide a consistent mapping between bags and target outputs throughout the entire procedure.

Each bag is then evaluated using a bag-level scoring function $F(B_v)$, defined as a weighted sum of the individual node-level predictions $f(u)$ over the instances $u \in B_v$:

[13]The target table contains the entities for which we want to make predictions. For example, in a *driver position* prediction task for Formula 1 pilots each target instance would be a specific driver for which we want to predict the final race ranking.

$$F(B_v) = \sum_{u \in B_v} \alpha(u, B_v) \cdot f(u).$$

Here, $f(u)$ is a trainable scoring function that estimates how indicative a single node u is for the target label, while $\alpha(u, B_v)$ is a trainable attention weight expressing the relative importance of node u within its bag. Intuitively, $f(u)$ measures the degree to which the presence of node u supports the prediction of the target label for the original target node v, and $\alpha(u, B_v)$ acts as a normalization or prioritization factor, ensuring that more informative nodes contribute more strongly to the final bag score.

For example, in a *driver position* prediction task, suppose B_v is the bag corresponding to a specific driver, and the bag contains all races in which this driver has participated in the current season. Here, $f(u)$ might estimate how much a particular race result u suggests that the driver will finish in the top 3 in the upcoming race. If a certain race outcome is highly predictive (e.g., a first-place finish against strong competitors), the model will learn a higher $f(u)$ and $\alpha(u, B_v)$ for that node, increasing its impact on $F(B_v)$. Less relevant races (e.g., races with incomplete data or low competition) will receive lower attention weights and/or lower scores.

Ideally, the discriminant function $F(\cdot)$ should separate the classes such that for any positive bag $B^+ \in S^+$ and any negative bag $B^- \in S^-$, we have:

$$F(B^+) > F(B^-).$$

In other words, all positive bags should have a higher aggregated score than all negative bags. This problem would be trivial, for example, in the case where every positive bag contains a node v that has an r-successor which is not also an r-successor of any node in a negative bag. In that scenario, assigning a weight of 1 to all such distinctive r-successor nodes, and a weight of 0 to all others, would perfectly separate the classes.

In realistic settings, however, the complex connectivity patterns of relational data preclude such simple solutions. Relations and node occurrences often overlap between positive and negative bags, noisy patterns emerge, and perfect separation is generally impossible. Therefore, the model must learn to combine multiple partially informative signals, weighting each according to its estimated contribution to the target prediction, rather than relying on a single decisive node or relation. The model is trained to assign higher scores to positive bags than to negative ones by minimizing a pairwise ranking loss:

$$\mathcal{L} = \sum_{B^+ \in \mathcal{S}^+, B^- \in \mathcal{S}^-} \sigma\left(F(B^-) - F(B^+)\right)$$

where σ is the sigmoid function, and $\mathcal{S}^+$ and $\mathcal{S}^-$ are the sets of positive and negative bags respectively.

After computing the classification loss for each candidate extension r, the relation with the lowest validation loss is selected to extend the current meta-path:

$$r^* = \arg\min_{r \in \mathcal{R}} \min_{\Theta, w} \mathcal{L}(r, \Theta, w)$$

This process is repeated iteratively until a maximum meta-path length is reached or the loss no longer improves significantly. At each step, the bags are updated by recursively propagating node-level importance weights α from the previous iteration to the current one, thereby maintaining the interpretability of the constructed path.

Let score($\mathcal{M}$) be the validation performance of a lightweight GNN trained on the semantic views induced by meta-paths in $\mathcal{M}$. At each iteration k, the algorithm selects the meta-path $\mathcal{P}^*$ that provides the maximum marginal improvement:

$$\mathcal{P}^* = \arg\max_{\mathcal{P} \in \mathcal{C} \setminus \mathcal{M}} \text{score}(\mathcal{M} \cup \{\mathcal{P}\})$$

and updates the selected set:

$$\mathcal{M} \leftarrow \mathcal{M} \cup \{\mathcal{P}^*\}$$

until $|\mathcal{M}| = K$.

The algorithm also includes an additional stopping criterion based on the marginal improvement that a candidate relation r brings to the prediction score. In practice, if appending a new relation to the current meta-path does not improve performance beyond a predefined threshold[14], the meta-path selection process terminates.

The process is detailed in the following Algorithm:

[14]In the original paper, this threshold is set to 30% of the previous performance, meaning that the newly extended meta-path must achieve at least 30% of the improvement obtained in the previous step in order to be considered.

Algorithm 1 MPS-GNN Learning

procedure LEARNMPS-GNN($\mathcal{G}, \mathcal{R}, \mathcal{Y}, L_{\text{MAX}}, \eta$)
  $mp^* \leftarrow [\,],\ F_1^* \leftarrow 0,\ S \leftarrow \mathcal{Y},\ A \leftarrow 1$
  while $|mp| < L_{\text{MAX}}$ do
    $r^* \leftarrow \arg\min_r L(r)$
    if $\min_{r \in \mathcal{R}} L(r) \geq \eta_{\text{Init}}(r)$ then
      return mp^*
    end if
    $mp \leftarrow mp + r^*$
    $gnn \leftarrow$ TRAIN(MPS-GNN(mp), $\mathcal{G}, \mathcal{Y}$)
    $F_1 \leftarrow$ TEST(gnn)
    if $F_1 > F_1^*$ then
      $mp^* \leftarrow mp,\ F_1^* \leftarrow F_1$
    end if
    $\mathcal{A}, S \leftarrow$ NEW-TARGETS(S, r^*)
  end while
  return mp^*
end procedure

The algorithm illustrated above outlines the complete MPS-GNN procedure for the single meta-path case. In the full version, however, the model selects more than one meta-path. This is achieved by employing a beam search strategy over the learned meta-paths, ranking them according to their predictive performance and retaining the top candidates as a final step.

The algorithm takes as input:

- $\mathcal{G}$: the input graph;
- $\mathcal{R}$: the set of edge types present in the graph;
- $\mathcal{Y}$: the set of labels for the target nodes.
- L_{max}: the maximum number of relations (edge types) allowed in a meta-path;
- η: the improvement threshold;

The procedure starts with an empty meta-path and initializes the set of targets $\mathcal{S}$ to contain the target nodes, each in its own singleton bag, along with their associated attention values $\mathcal{A}$ set to 1. At each iteration, all possible one-step extensions of the current meta-path are considered by appending a candidate relation $r \in \mathcal{R}$. The candidate that minimizes the loss function $L(r)$ is selected and temporarily added to the meta-path. If no relation achieves an improvement

greater than the threshold η, the search process terminates and the current best meta-path is returned.

Whenever a relation is appended, the MPS-GNN is trained[15] (through the execution Train(MPS-GNN(mp), G, Y)) using the extended meta-path to predict the target labels $\mathcal{Y}$ from the graph $\mathcal{G}$, and its performance is evaluated on a validation set using the F_1 score. If this score surpasses the best score obtained so far, the current meta-path and its score are stored as the new best meta-path. The set of target bags $\mathcal{S}$ and attention values $\mathcal{A}$ are then updated.

Finally, the embeddings produced by each of the K selected meta-paths are concatenated to form the final representation of node v at layer $(l+1)$:

$$h_v^{(l+1)} = \Big\|_{k=1}^{K} h_{(v,k)}^{(l+1)}, \tag{2.6.1}$$

where $h_{(v,k)}^{(l+1)}$ denotes the embedding of node v obtained by aggregating information along the k-th meta-path, and $\|$ represents the concatenation operator. This concatenated representation is then passed to the subsequent layers of the model or directly to the prediction layer, allowing the GNN to jointly exploit multiple complementary relational patterns.

This approach allows the model to exploit multiple complementary relational patterns while keeping the search space tractable. Note that a naive search algorithm that simply tests all possible meta-paths would have a polinomial compexity, making it impossible to be used for a big amount of edge types.

This technique, instead, by leveraging the scoring function, reduces the time complexity to liear $O(|R| \cdot L)$. *Here, $|R|$ denotes the number of relation types in the graph, and L denotes the maximum meta path length (number of hops)*[16].

Once the meta-paths have been automatically constructed, each meta-path $r_1, \ldots, r_L$ defines a multi-relational GNN with L layers, where each layer corresponds to one specific relation in the meta-path. The first layer processes the initial node type of the meta-path, and the computation proceeds sequentially until the final relation r_L is applied.

For a single meta-path, the forward propagation from layer l to layer $l+1$ for a

[15]Note that a weighted multi-instance classification approach is employed to both improve predictive performance and accelerate the meta-path selection process. Without this mechanism, the MPS-GNN model would need to exhaustively evaluate every possible relation, resulting in significantly higher computational cost.

[16]As mentioned in[50], this improvement is due to the fact that at each step, one relationship is added to the meta-path under construction.

node v is given by:

$$h_v^{(l+1)} = \sigma\left(W_0^{(l)} h_v^{(l)} + W_{\text{neigh}}^{(l)} \sum_{u \in \mathcal{N}_v^{r_{L-l}}} h_u^{(l)} + W_1^{(l)} h_v^{(0)}\right), \tag{2.6.2}$$

where $\mathcal{N}_v^{r_{L-l}}$ denotes the set of neighbors of node v connected through the relation r_{L-l}, $h_v^{(l)}$ is the embedding of node v at layer l, and $h_v^{(0)} = x_v$ is the initial feature vector of node v. The matrices $W_0^{(l)}$, $W_{\text{neigh}}^{(l)}$ and $W_1^{(l)}$ are learnable parameters, while $\sigma(\cdot)$ denotes a non-linear activation function.

The term $W_1^{(l)} h_v^{(0)}$ acts as a skip connection from the input features to the $(l+1)$-th layer, allowing the model to directly access the original node attributes at every step. This design choice is crucial for enabling the MPS-GNN to capture node-level information corresponding to the statistical terms computed during meta-path construction.

When multiple meta-paths are selected (as in the beam search setting), the embeddings obtained from each meta-path are concatenated to form the final node representation following equation(2.6.1). The resulting concatenated representation is then used in the final prediction layer.

A key advantage of MPS-GNN is that the entire meta-path selection and attention process is explicitly exposed.

This enables clear and detailed **explanations** of the relational reasoning performed by the model. Moreover, since the model operates in a fully differentiable and end-to-end fashion, it can be trained using standard backpropagation with supervision from a downstream task-specific loss.

Despite its strong performance in binary node classification tasks, MPS-GNN exhibits limitations when applied to more general settings. In particular, the meta-path selection process relies on a greedy evaluation strategy that is optimized for classification accuracy on **binary tasks**. This focus restricts its applicability in multi-class classification scenarios, where semantic diversity among classes may require a richer and more balanced set of meta-paths. More critically, MPS-GNN is not designed for node-level regression tasks, where the model must capture fine-grained continuous variations in target values rather than discrete class boundaries. In such cases, the discrete meta-path selection procedure and downstream aggregation mechanisms may fail to capture the nuanced structural dependencies required for accurate predictions.

As a result, despite being very effective in revealing salient relational semantics for classification, MPS-GNN limits its generalization to a broader range of tasks in relational learning.

Furthermore, while MPS-GNN learns attention weights to quantify the relative importance of each selected meta-path, it does not include a second-level attention mechanism to assess the contribution of individual intermediate paths within a

meta-path occurrence. This limits the granularity of interpretability to the meta-path level, making it less suited for scenarios where fine-grained, node-specific explanations are required. Finally, this approach lacks a formalized interpretability framework or systematic method for extracting human-readable explanations[17].

Enhancing Interpretability through LLMs

A promising line of research focuses on leveraging large language models (LLMs) to enhance interpretability by verbalizing meta-paths into natural language explanations. One notable work, Metapath of Thoughts: Verbalized Metapaths in Heterogeneous Graph as Contextual Augmentation to LLM [51], exemplifies this approach.

In this framework, meta-paths are first learned by training a FastGTN[47], which assigns importance weights to different high-order relationship patterns. The top-K meta-path types (starting from the target node type) are selected according to these learned weights.

Each meta-path type is verbalized via an LLM prompt template that encodes node types, relation semantics, and structure, producing a coherent natural-language description. For each target node, instances of the selected meta-path types are sampled proportionally to their importance weights, and each instance is transformed into a narrative prompt by filling the verbalized template with the specific nodes, edges, and their textual attributes.

Additionally, a small set of semantically similar, labeled nodes is retrieved to serve as in-context examples. The final input to the LLM includes the target node's textual attributes, the collection of verbalized meta-path instances, and the retrieved examples, followed by a structured reasoning prompt that elicits both a prediction and a human-readable explanation.

This approach shows how LLM calls can turn structural meta-path semantics into intuitive, context-rich narratives, offering clearer interpretability than traditional saliency or attention-based explainers and pointing toward a powerful integration of graph structure learning and language-based reasoning in relational deep learning.

Example: In a bibliographic network, instead of showing the symbolic meta-path `P-A-P` (paper–author–paper), the model verbalizes it as:

> "The target paper shares an author with Paper X, which discusses deep learning on heterogeneous graphs."

[17] In fact, even if MPS-GNN inherently exposes interpretable components, such as the selected meta-paths, it does not provide a user-friendly explanation that allows any user to understand the reasoning process adopted by the model. It only provides the meta-paths that were used to make the predictions, but without a verbalized and clear reason about why the model performed a certain prediction.

and pairs it with additional paths such as:

> "The target paper is published in the same conference as Paper Y, which focuses on graph representation learning."

By embedding these narrative instances along with the textual attributes of the connected papers, the explanation becomes accessible and meaningful even to users without prior knowledge of graph notation. This narrative, example-driven format offers a **clearer and more relatable justification** of the prediction than numeric attention weights or symbolic meta-paths alone, making the model's reasoning process transparent to both technical and non-technical stakeholders.

2.7 End-to-End Modelling in RelBench

The RelBench benchmark[4, 9] is more than just a collection of datasets; it is also a *reference pipeline* that demonstrates how modern graph representation learning can be executed *directly* on relational schemas without flattening or manual propositionalisation [4], by using Graph Neural Networks over temporal and heterogeneous graphs. This section gives a technical walk-through of that pipeline and places it in the context of earlier work on relational deep learning.

2.7.1 Graph construction from normalised schemas

Starting from a relational database, RelBench builds a heterogeneous graph that preserves **all** information in the original schema: no rows, columns, or relationships are lost.

The result of this transformation is a **heterogeneous, attributed, and temporal graph** that faithfully mirrors the original relational database structure. Each node belongs to a specific type corresponding to its source table, and each edge encodes a primary–foreign key relationship.

Concretely, each row in every table becomes a node whose *type* is the name of that table. This design allows different node types to carry distinct attribute sets, mirroring the column layouts of their source tables. Every primary–foreign-key constraint is converted into a *typed, directed edge*. Multiple constraints therefore yield multiple edge types, each encoding a different semantic relation.

Formally, the output graph G consists of:

- a node set $V = \bigsqcup_{\tau \in \mathcal{T}} V_\tau$, partitioned by node type;
- an edge set $E = \bigsqcup_{r \in \mathcal{R}} E_r$, where each E_r corresponds to a distinct relation type (i.e., foreign-key constraint);

- node attributes stored in a dictionary `x_dict`, where each entry maps a node type to its feature matrix;
- edge connectivity stored in `edge_index_dict`, a dictionary mapping each edge type to a tensor of shape [2, `num_edges`];

The entire transformation is *deterministic*, *schema-agnostic*, and runs in time linear in the number of tuples, guaranteeing reproducibility across organisations and hardware.

The runtime scales linearly with the number of tuples, and the output graph is *byte-for-byte reproducible*: two independent users running the converter on the same dump will obtain identical node IDs, edge lists and feature matrices.

The entire transformation is **fully automated**: once a database dump and its schema are provided, no further human intervention is required. Because the algorithm relies solely on the formal definition of primary and foreign keys, and never on domain-specific heuristics, it works *unchanged* for any dataset and any task-type.

2.7.2 Feature encoding: *HeteroEncoder*

Given a set of TensorFrames[18] objects, one for each table of original schema, RelBench first applies **HeteroEncoder**. Each of the TensorFrames stores all nodes of a given type[19]. More specifically, it stores their raw feature columns, and the corresponding data-type (e.g., numerical, categorical, textual, temporal). HeteroEncoder converts these heterogeneous columns into a PyTorch tensor containing the embeddings for the nodes and returns a dictionary that stores a d-dimensional vector representation for every node[20] of every table. The result is a dictionary of node embeddings:

$$\texttt{x\_dict} \;=\; \Big\{\, \textit{node\_type} \;\mapsto\; \mathbf{X}_\tau \in \mathbb{R}^{N_\tau \times d} \Big\}, \tag{2.7.1}$$

This dictionary contains, for each node type[21], a tensor of shape $N_\tau \times d$. Here, N_τ denotes the number of nodes of type τ, and d is the shared embedding dimension

[18] **TensorFrame** is a data structure introduced in *PyTorch Frame* [52] designed to represent heterogeneous tabular data in a tensor-friendly format for deep learning. It stores raw column values together with their semantic types (e.g., numerical, categorical, textual, temporal) and supports efficient *materialization* into PyTorch tensors, enabling column-wise encoding and interaction before aggregation into dense row embeddings.

[19] We can also say that it stores all the records of a specific table of the relational schema.

[20] Notice that each node is an instance of the relational database.

[21] Corresponding to the name of the table from which the instances originate

across all node types. Each row of this tensor represents a single node instance, and each row vector of length d stores the learned embedding obtained by encoding and aggregating its original attributes.

For every node type τ, columns are partitioned by semantic type: $\mathcal{C}_\tau = \bigsqcup_{s \in \mathcal{S}} \mathcal{C}_{\tau,s}$. Each semantic type s has an encoder E_s applied column-wise and row-wise.

For each row (node) i, the per-column embeddings are concatenated and fed to a small residual MLP, denoted "ResNet" in TorchFrame[22] applied row-wise, which produces a d-dimensional node embedding:

$$\left[\mathbf{z}_i^{(c_1)} \,\|\, \mathbf{z}_i^{(c_2)} \,\|\, \cdots\right] \xrightarrow{\phi_\tau} \mathbf{x}_i^{(\tau)} \in \mathbb{R}^d, \qquad i = 1, \ldots, N_\tau,$$

$$\text{with} \quad \mathbf{z}_i^{(c)} \in \mathbb{R}^{d_c} \text{ for } c \in \mathcal{C}_\tau.$$

where ϕ_τ denotes the per-type fusion network (e.g., residual blocks + LayerNorm/activation + final linear).

Figure 2.10: Overview of the RelBench pipeline. The process starts from a relational **database**, from which features are extracted and encoded (e.g., using *GloVe* for text or categorical embeddings). Temporal information is separately encoded and combined with the original feature space. The result is a **heterogeneous graph**, where each node represents an entity and edges represent typed relations (including timestamps, when available). This graph is then processed by a Graph Neural Network (GNN) architecture, followed by an MLP head that performs prediction (classification, regression, or recommendation) on the target nodes.

2.7.3 TemporalHeteroEncoder: Injecting Time into Node Features

Relational tasks in RelBench are explicitly temporal: each training instance is associated with a *prediction time* t_{pred}. To enable the GNN to exploit temporal informations, **TemporalHeteroEncoder** augments the static node embeddings produced by *HeteroEncoder* with *relative* time information.

[22]Here, "ResNet" means a row-wise fully connected residual network for tabular features (Linear–Norm–ReLU–Linear with a skip connection), not the convolutional ResNet for images.

At run time it receives three inputs: (i) a vector of reference times for the T seed nodes, $\texttt{seed\_time} \in \mathbb{R}^T$; (ii) a dictionary of per-node timestamps, $\texttt{time\_dict} = \{\tau \mapsto \mathbf{t}_\tau \in \mathbb{R}^{N_\tau}\}$; (iii) a dictionary of seed assignments, $\texttt{batch\_dict} = \{\tau \mapsto \mathbf{b}_\tau \in \{0, \dots, T-1\}^{N_\tau}\}$, which maps each sampled node to the index of its owning seed in the current mini-batch. The encoder outputs

$$\texttt{rel\_time\_dict} \;=\; \left\{\tau \mapsto \mathbf{R}_\tau \in \mathbb{R}^{N_\tau \times d}\right\}_{\tau \in \mathcal{T}_V^{\text{time}}}.$$

For a node u of type τ, let $s = \mathbf{b}_\tau[u]$ be its seed index, $t^{\text{ref}} = \texttt{seed\_time}[s]$ the corresponding reference time, and $t_u = \mathbf{t}_\tau[u]$ its timestamp. The encoder computes a time-based vector $\mathbf{r}_u = h_\tau(t^{\text{ref}}, t_u) \in \mathbb{R}^d$ and sets the u-th row of $\mathbf{R}_\tau$ equal to $\mathbf{r}_u$; the map h_τ is implemented inside the module and is designed to produce d-dimensional representations aligned with the static embeddings. In the model, these temporal representations are added elementwise to the outputs of the **HeteroEncoder**:

$$\mathbf{X}_\tau \leftarrow \mathbf{X}_\tau + \mathbf{R}_\tau \qquad (\tau \in \mathcal{T}_V^{\text{time}}),$$

so that each node embedding incorporates its time information while preserving shape $\mathbb{R}^{N_\tau \times d}$ and the interface expected by the downstream heterogeneous GNN. Node types not in $\mathcal{T}_V^{\text{time}}$ do not appear in `rel_time_dict` and their embeddings remain unchanged.

2.7.4 Mini-batching with `NeighborLoader`

Relational graphs extracted from enterprise databases typically contain hundreds of thousands of nodes and millions of edges. Training GNNs on the entire graph at once is computationally infeasible and unnecessary. For this reason, RelBench adopts a mini-batch training strategy based on the `NeighborLoader` module from PyTorch Geometric. Performing mini batch sampling on graph is hard, especially when dealing with temporal instances. In this context we need to avoid any possible **data leakage**, which means considering future instances durign training and testing when making prediction on previous instances.

The `NeighborLoader` performs layer-wise neighborhood sampling during training. Given a set of seed nodes[23] for a prediction batch, the loader recursively samples a fixed number of neighbors at each hop, constructing a **subgraph around the seed nodes** that can be processed efficiently by the model. The number of neighbors per hop can be configured as a list (e.g., $[15, 10]$[24]), where each element corresponds to the maximum number of neighbors sampled at that layer.

[23] We call 'seed nodes' the nodes for which we want to make a prediction.

[24] That is, we sample up to 15 first-hop neighbors and up to 10 second-hop neighbors.

By default, RelBench uses uniform sampling in the `NeighborLoader`. In this sampling strategy, at each hop, and for each seed (or frontier) node, it first collects the set of eligible neighbors (respecting any temporal constraint, e.g. neighbors whose timestamp is not later than the reference time of the seed). From this candidate set, it then draws up to the requested budget of neighbors *at random and without replacement*, giving every candidate the same chance of being selected. If fewer neighbors are available than the requested ones, all available neighbors are taken. In heterogeneous graphs, the same rule is applied independently for each edge type. The procedure is repeated hop by hop, and the union of the sampled nodes and edges forms the mini-subgraph for the current batch. For example, with `num_neighbors = [15, 10]` and `temporal_strategy = "uniform"`, the loader picks up to 15 first-hop and up to 10 second-hop neighbors uniformly at random among those that satisfy the temporal constraint.

The loader returns, at each iteration, a mini-batch subgraph $\mathcal{B}$ as a `HeteroData` object. Given a sampling budget `num_neighbors` for H hops (a list of length H, or, in heterogeneous graphs, a dictionary keyed by edge type), the loader iteratively expands the frontier: at hop $h \in \{1, \ldots, H\}$ it selects up to the prescribed number of neighbors for every frontier node, according to the chosen strategy (e.g., uniform or most-recent in temporal mode), and forms the induced mini-subgraph on the union of seeds and sampled neighbors. The returned $\mathcal{B}$ contains all node/edge attributes restricted to the sampled indices and uses *local* indexing. Only neighbors that satisfy the temporal constraint (e.g., timestamp not later than a reference time) are eligible at each hop

Among other data structures, the `HeteroData` object contain:

- `tf_dict`: A dictionary mapping each node type to a `TensorFrame` containing the raw features of the sampled nodes.
- `edge_index_dict`: A dictionary mapping each edge type to its sampled edges.
- `batch_dict`: A mapping from sampled nodes to their corresponding seed node index.
- `time_dict`: A timestamp vector for each node type, used for temporal encoding.

This approach enables scalable training across heterogeneous and temporal graphs, while maintaining compatibility with modular encoders like the `HeteroEncoder` and `HeteroTemporalEncoder`. It supports arbitrary relational schemas and large-scale datasets without manual engineering.

As a concrete example, we instantiate a separate `NeighborLoader` for each split (`train`, `val`, `test`) as follows:

```
loader_dict[split] = NeighborLoader(
    data,
    num_neighbors=[num_neighbours for _ in range(2)],
    input_nodes=table_input.nodes,
    input_time=table_input.time,
    transform=table_input.transform,
    batch_size=batch_size,
    temporal_strategy="uniform",
    shuffle=split == "train",
    num_workers=0,
    persistent_workers=False,
)
```

Each argument plays a crucial role in constructing efficient and correct subgraphs for mini-batch training:

- `data`: The full `HeteroData` object representing the entire heterogeneous and temporal graph extracted from the relational database.
- `num_neighbors`: A list specifying the number of neighbors to sample per layer. For example, `[15, 10]` samples up to 15 neighbors at the first hop, and 10 at the second. Here, a constant value `num_neighbours` is broadcast to both layers, enabling uniform sampling depth.
- `input_nodes`: A tuple `(node_type, node_indices)` that identifies the target nodes for the prediction task. These nodes act as seeds from which neighborhoods are sampled.
- `input_time`: A vector $\mathbf{t}_{\text{seed}} \in \mathbb{R}^B$ specifying the prediction time associated with each seed node. It is used to apply temporal constraints to the sampled neighbors (i.e., only neighbors with timestamps $t < t_{\text{seed}}$ are considered valid).
- `transform`: A callable used to apply custom transformations (e.g., feature preprocessing, timestamp normalization) to the sampled subgraph before it is passed to the model.
- `batch_size`: The number of seed nodes sampled per mini-batch. This controls the overall subgraph size and training throughput.
- `temporal_strategy`: Specifies the strategy for time-aware neighborhood sampling. `"uniform"` applies uniform sampling among temporally valid neighbors.
- `shuffle`: When set to `True`, the seed nodes are shuffled before batching. This is typically enabled during training and disabled for evaluation.

- `num_workers` Controls how many subprocesses the underlying PyTorch `DataLoader` uses to load and collate mini-batches. A value of `0` performs all loading work in the main process (simpler, lower memory overhead, but single-threaded I/O and preprocessing). A value `num_workers` > 0 spawns that many worker processes, enabling parallel fetching, decoding, and transformations, which can increase throughput at the cost of higher RAM usage and inter-process communication overhead.

- `persistent_workers`: If set to `True` (and only effective when `num_workers` $>$ 0), worker processes are kept alive across successive iterations/epochs, avoiding the startup/teardown cost at every epoch and thus reducing latency in long trainings. If `False`, workers are shut down when the iterator is exhausted. This flag is useful when data loading/transforms are non-trivial and the loader is re-iterated many times; it has no effect when `num_workers` $= 0$.

By controlling both the relational neighborhood and the temporal constraints, the `NeighborLoader` enables efficient and causally consistent training over dynamic, relational data structures without the need to load the entire graph in memory.

2.7.5 Message-passing backbone

The core component of the RelBench pipeline is a relational Graph Neural Network tailored to heterogeneous graphs. In particular, the default architecture employs a multi-layer variant of *GraphSAGE* [25] adapted to relational domains. The resulting module, called `HeteroGraphSAGE`, supports distinct message-passing schemes for each edge type and incorporates layer normalization and non-linearities per node type.

Each layer in the `HeteroGraphSAGE` stack applies an update rule for the k-th layer, which is given by:

$$h_v^{(k)} = \sigma\left(W_{\text{self}}\, h_v^{(k-1)} \; + \sum_{r=(\tau_s \to \tau_t)} \frac{1}{|\mathcal{N}_r(v)|} \sum_{u \in \mathcal{N}_r(v)} W_r\, h_u^{(k-1)} \right),$$

where:

- $\mathcal{N}_r(v)$ denotes the set of r-type neighbors of node v.
- $W_r \in \mathbb{R}^{d \times d}$ is the relation-specific weight matrix for edge type r.
- $W_{\text{self}} \in \mathbb{R}^{d \times d}$ is the shared self-loop transformation.
- σ is a non-linear activation function, typically ReLU.

This design allows different edge types to contribute differently to the representation of a given node, capturing heterogeneous semantics in the neighborhood structure.

After the aggregation step, each node embedding is passed through a `LayerNorm` module applied per node type, followed by a ReLU activation. Formally:

$$h_v^{(k)} \leftarrow \mathrm{ReLU}(\mathrm{LayerNorm}_\tau(h_v^{(k)})),$$

where τ is the type of node v and the normalization parameters are learned independently for each type. This design helps stabilize training and mitigate internal covariate shift across different node domains.

2.7.6 Prediction head and loss functions

The final node embeddings $h_v^{(L)} \in \mathbb{R}^d$ produced by the GNN backbone are fed to a task-specific *prediction head* implemented as an MLP, whose architecture and output dimensionality depend on the task. For example, in **binary classification** we set $\mathrm{MLP}_\theta : \mathbb{R}^d \to \mathbb{R}$ and apply a sigmoid σ to obtain $p(y{=}1 \mid v) = \sigma(\mathrm{MLP}_\theta(h_v^{(L)}))$; in **multi-class classification** we use $\mathrm{MLP}_\theta : \mathbb{R}^d \to \mathbb{R}^C$ followed by a softmax over C classes; in **single-target regression** the head maps $\mathbb{R}^d \to \mathbb{R}$; in **multi-task regression** it maps $\mathbb{R}^d \to \mathbb{R}^k$ for k targets; and in **recommendation** it outputs a real-valued score per item (e.g., $\hat{r}_{v,i} \in \mathbb{R}$ for ranking).

The appropriate loss function and evaluation metric are automatically selected by the RelBench framework based on the task specification.

2.7.7 Strengths, limitations, and outlook

The pipeline operates on heterogeneous temporal graphs using `HeteroData` and tabular TensorFrames, and requires only minimal task-specific glue. Temporal neighbor sampling (`time_attr`, optional `input_time`, and a temporal strategy) enforces causal correctness by admitting only time-consistent neighbors at each hop. The design is modular: static encoders (*HeteroEncoder*), temporal encoders (*HeteroTemporalEncoder*), the sampler (*NeighborLoader*), the GNN backbone (e.g., *HeteroGraphSAGE*), and the prediction head are separable components and can be swapped or ablated independently. Subgraph mini-batching with multi-hop sampling allows training on graphs that are too large for full-batch processing, while still exposing a controllable receptive field around seed nodes.

However, nodes without incident edges in the sampled subgraph (or with very sparse neighborhoods under the chosen sampling budget) contribute little relational signal and are represented mainly by their tabular encodings; performance on such "cold-start" cases therefore hinges on feature quality and the static encoder. This

is not a limitation of the pipeline; rather, it is a scenario in which relational deep learning tends to underperform.

The behavior and cost of the method depend materially on sampler hyperparameters (number of hops, per-hop budgets, and the temporal strategy): small budgets can truncate useful context, whereas large budgets increase memory/latency and may introduce more temporal staleness and model's design space (type of GNN, number of layers, etc.).

Chapter 3

Proposed Method

As relational data form the backbone of modern data infrastructure, establishing a strong baseline for this setting is more than an academic exercise: it is a lever that can translate directly into efficient, automated solutions to many real-world problems. By casting relational databases into heterogeneous temporal graphs, relational deep learning provides a flexible foundation with significant potential. Yet it remains an emerging, innovative field with boundaries only beginning to take shape and potential still largely unexplored.

A wide range of choices, including temporal encoding, pre-training protocols, GNN architectures, and evaluation design, remains unexplored to develop a robust, reproducible solution for node-level tasks. Thus, we conducted experiments to enhance various aspects of the current pipeline.

3.1 Model selection

As a first step toward that goal, we focus on the architectural backbone of the GNN, examining how message-passing mechanisms shape performance.

We chose to begin by exploring the model design, as it is the most influential and impactful architectural decision, shaping the model's inductive bias and capacity, and providing a stable foundation for comparing subsequent choices. Once this backbone is clarified, we can systematically investigate complementary dimensions.

As discussed, RelBench proposes the use of a modified version of the GraphSAGE model[25] designed to work on Heterogeneous graphs; however, this model is relatively simple, and it is reasonable to assume that alternative architectures, which have shown superior performance in other contexts, could yield better results in this one as well.

3.1.1 Heterogeneous Graph Attention Network

Our implementation of the heterogeneous GAT model extends the idea of Graph Attention Network to heterogeneous temporal graphs by assigning a distinct `GATConv` layer to each relation type in the graph. Formally, let $\mathcal{T}$ denote the set of node types and $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{L} \times \mathcal{T}$ the set of edge types, where $\mathcal{L}$ denotes the set of relation labels. For each edge type $r = (s, rel, d) \in \mathcal{R}$, we maintain an independent GAT[1] module with parameters $\mathbf{W}^{(r)}$ and attention vector $\mathbf{a}^{(r)}$.

Given input node embeddings $\mathbf{X}^{(s)} \in \mathbb{R}^{n_s \times F_s}$ and $\mathbf{X}^{(d)} \in \mathbb{R}^{n_d \times F_d}$, the GAT associated with relation r computes messages as:

$$h_j^{(r)} = \sigma\left(\sum_{i \in \mathcal{N}^{(r)}(j)} \alpha_{ij}^{(r)} \mathbf{W}^{(r)} \mathbf{x}_i\right),$$

where $\mathcal{N}^{(r)}(j)$ denotes the neighbors of node j under relation r, σ is a non-linear activation, and the attention coefficients $\alpha_{ij}^{(r)}$ are defined as:

$$\alpha_{ij}^{(r)} = \frac{\exp\left(e_{ij}^{(r)}\right)}{\sum_{k \in \mathcal{N}_r(j)} \exp\left(e_{kj}^{(r)}\right)},$$

Where $e_{ij}^{(r)}$ is the raw score, defined as:

$$e_{ij}^{(r)} = \text{LeakyReLU}\Big((a^{(r)})^\top [\, W_s^{(r)} x_i \,\|\, W_d^{(r)} x_j \,]\Big),$$

In practice, $\alpha_{ij}^{(r)}$ is the attention weight assigned to neighbor i when aggregating into node j for relation r. It is a softmax over the raw scores $e_{ij}^{(r)}$ computed only across the neighbors of j under relation r, so $\sum_{i \in \mathcal{N}_r(j)} \alpha_{ij}^{(r)} = 1$.

Here, $a^{(r)}$ is the learnable attention vector for relation r, $W_s^{(r)}$ and $W_d^{(r)}$ are the linear projections for source and destination node features, x_i and x_j are the input features, $\mathcal{N}_r(j)$ is the neighbor set of j under relation r, and $\|$ denotes concatenation.

We can also write:

$$\alpha_{ij}^{(r)} = \frac{\exp\Big(\text{LeakyReLU}\Big((a^{(r)})^\top [\, W_s^{(r)} x_i \,\|\, W_d^{(r)} x_j \,]\Big)\Big)}{\sum_{k \in \mathcal{N}_r(j)} \exp\Big(\text{LeakyReLU}\Big((a^{(r)})^\top [\, W_s^{(r)} x_k \,\|\, W_d^{(r)} x_j \,]\Big)\Big)}.$$

[1] We rely on the standard implementation of GATConv provided by PyTorch Geometric, which follows the original GAT formulation[27].

Since a node type d may receive incoming messages from multiple relations $\mathcal{R}_d = \{(s, rel, d) \in \mathcal{R}\}$, we employ the `HeteroConv` operator to aggregate the outputs of each relation-specific GAT. For node $j \in \mathcal{V}_d$, the aggregated representation is:

$$h_j = \text{AGGREGATE}\left(\{h_j^{(r)} \mid r \in \mathcal{R}_d\}\right),$$

where `AGGREGATE` denotes a chosen combination function (e.g., sum, mean, max, or concatenation)[2].

Stacking multiple layers of such relation-aware attention allows nodes to iteratively refine their embeddings by integrating information from multi-relational neighborhoods, while keeping the attention mechanism specific to each edge type.

This approach is fundamentally identical to the one adopted in the RelBench baseline [4, 9], with the only difference being that we employ `GATConv` layers instead of `SAGEConv`[3].

This design allows us to extend the original GAT[27] to heterogeneous graphs. We have already discussed an extension to heterogeneous graphs known as HAN[33]; in contrast, our approach does not rely on metapaths, which are a common bottleneck because they typically require metapaths crafted by domain experts.

The advantage of this implementation over a standard HAN [33] model lies in the fact that, in our case, we do not need to rely on hand-crafted metapaths.

3.1.2 Heterogeneous Graphormer

The Transformer [26] has established itself as one of the most powerful neural architectures for modeling sequential data. In recent years, several efforts have been made to extend the Transformer model to the graph domain, among which the Graphormer architecture [32] has emerged as one of the most promising approaches.

Graphormer is directly built upon the standard Transformer model and achieve state-of-the-art performances on homogeneous graph structures.

To appreciate the innovative design of Graphormer, we first need a solid understanding of the Transformer architecture.

The Transformer Architecture

In this section, we do not focus on the entire Transformer model, but mainly on its encoder component, which is the part most relevant to our analysis.

The Transformer [26] is a neural architecture designed to model sequential data by relying on attention mechanisms, dispensing with recurrence and convolution.

[2] In our experiments we used the max function.

[3] Which follow the definition of GraphSAGE as introduced in the original paper [25]

Its core building block is the **self-attention** mechanism, which enables each token in a sequence to directly attend to all others.

Given a sequence of n tokens $(x_1, x_2, \ldots, x_n)$, each token is embedded into a d_{model}-dimensional vector:

$$\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n] \in \mathbb{R}^{n \times d_{\text{model}}}.$$

Since Transformers were originally developed for natural language processing, where token order is crucial, positional information is injected by adding a **positional encoding** $\mathbf{p}_i$ to each token embedding $\mathbf{x}_i$(this makes the Transformer architecture **non–permutation-invariant**):

$$\mathbf{z}_i = \mathbf{x}_i + \mathbf{p}_i, \qquad \mathbf{Z} = [\,\mathbf{z}_1, \ldots, \mathbf{z}_n\,].$$

Each token embedding $\mathbf{z}_i \in \mathbb{R}^{d_{\text{model}}}$ is linearly projected into the *query*, *key*, and *value* spaces via learned projection matrices $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_{qkv}}$:

$$\mathbf{q}_i = \mathbf{z}_i \mathbf{W}^Q, \qquad \mathbf{k}_i = \mathbf{z}_i \mathbf{W}^K, \qquad \mathbf{v}_i = \mathbf{z}_i \mathbf{W}^V.$$

Stacking all tokens gives the matrices $\mathbf{Q} = \mathbf{Z}\mathbf{W}^Q$, $\mathbf{K} = \mathbf{Z}\mathbf{W}^K$, and $\mathbf{V} = \mathbf{Z}\mathbf{W}^V$, with $\mathbf{Z} = [\,\mathbf{z}_1, \ldots, \mathbf{z}_n\,] \in \mathbb{R}^{n \times d_{\text{model}}}$.

with $\mathbf{W}^Q, \mathbf{W}^K, \mathbf{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_k}$.

The unnormalized attention score between token i and token j is:

$$\alpha_{ij} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j^\top}{\sqrt{d_k}}.$$

These scores are normalized using the softmax function:

$$a_{ij} = \frac{\exp(\alpha_{ij})}{\sum_{j'=1}^{n} \exp(\alpha_{ij'})}.$$

The attention output for token i is then:

$$\mathbf{h}_i = \sum_{j=1}^{n} a_{ij} \mathbf{v}_j.$$

In matrix form, for all tokens:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right)\mathbf{V}.$$

Instead of a single attention function, H attention "heads" are computed in parallel:

$$\text{head}_h = \text{Attention}(\mathbf{Q}\mathbf{W}_h^Q, \mathbf{K}\mathbf{W}_h^K, \mathbf{V}\mathbf{W}_h^V).$$

The outputs are concatenated and projected:

$$\mathrm{MHA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_H)\mathbf{W}^O.$$

where W^{O} is the output projection matrix that maps the concatenated attention heads back to the model dimension.

This allows the model to capture different types of dependencies simultaneously.

Each position is independently passed through a two-layer feedforward network:

$$\mathrm{FFN}(\mathbf{z}) = \max(0, \mathbf{z}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2.$$

An encoder block consists of:

$$\mathbf{Z}' = \mathrm{LayerNorm}\Big(\mathbf{Z} + \mathrm{MHA}(\mathbf{Z}, \mathbf{Z}, \mathbf{Z})\Big),$$

$$\mathbf{Z}'' = \mathrm{LayerNorm}\Big(\mathbf{Z}' + \mathrm{FFN}(\mathbf{Z}')\Big).$$

Thus, residual connections and normalization stabilize training.

The Transformer replaces recurrence with global self-attention, enabling parallelization and effective modeling of long-range dependencies. Multi-head attention enriches representational capacity, while residual connections and normalization ensure stable training.

The Graphormer architecture

The Transformer has become a central architecture in modern machine learning, yet its native formulation assumes an ordered sequence and lacks an explicit notion of graph topology, which prevents a direct application to graphs; this motivates architectures that adapt attention to the irregular, permutation invariant structure of graphs, such as Graphormer.

Graphormer modifies the self-attention mechanism by integrating structural properties of the graph, thereby accounting for node connectivity.

Let $G = (V, E)$ be a graph with $|V| = n$ nodes. We denote the hidden representation at layer ℓ as $h_i^{(\ell)} \in \mathbb{R}^d$ and collect them row-wise into $H^{(\ell)} \in \mathbb{R}^{n \times d}$, with $H^{(0)} = X$ (the node's features).

Graphormer injects graph structure into the *attention logits* through three encodings: (i) Centrality encoding; (ii) Spatial encoding; (iii) Edge encoding.

The self-attention computation described in the Transformer architecture[26] does not account for the relative importance of each node. However, node importance is a key aspect in graphs, largely due to their scale-free nature [53]. As noted in [32], highly connected nodes (e.g. celebrities with a large number of followers) can be decisive in predicting the evolution of a social network. Among the many possible centrality measures, Graphormer adopts *degree centrality* and encodes it by

assigning to each node two real-valued embedding vectors: one for its indegree[4] and one for its outdegree[5]. These embeddings are simply added to the node features:

$$\mathbf{h}_i^{(0)} \;=\; \mathbf{x}_i \;+\; \mathbf{z}^-_{\deg^-(v_i)} \;+\; \mathbf{z}^+_{\deg^+(v_i)},$$

where $\mathbf{x}_i \in \mathbb{R}^d$ denotes the original node feature vector, and $\mathbf{z}_k^-, \mathbf{z}_k^+ \in \mathbb{R}^d$ are learnable embeddings indexed by the indegree and outdegree values, respectively[6].

As already discussed, a key advantage of the Transformer architecture is its use of *global* self-attention[7].

Graphormer adapts this mechanism to the graph domain by removing the Transformer's positional encoding to ensure **permutation invariance**, thus making the model suitable for graph-structured data. However, when computing attention scores between nodes, it is crucial to inform the model about their actual proximity in the graph. In other words, a **spatial encoding** mechanism is required to obtain well-calibrated attention scores over all node pairs by incorporating their relative graph distance. By doing so, the model downweights interactions between nodes that are distant or poorly connected, preventing spurious high attention on structurally unlikely pairs.

To encode the structural information of a graph, the authors of Graphormer introduce a spatial encoding mechanism. Specifically, they compute the shortest path distance (SPD) between every pair of nodes in the graph and inject this value as a bias term into the self-attention score. In this way, the attention mechanism is not only aware of global connectivity but can also distinguish how close two nodes are in the underlying graph structure.

The attention values are then computed as:

$$A_{ij} = \frac{\left(\mathbf{Q}_i \mathbf{K}_j^\top\right)}{\sqrt{d}} + b_{\phi(i,j)} \tag{3.1.1}$$

Where $b_{\phi(i,j)}$ is a learnable scalar indexed by $\phi(i,j)$ that contains the shortest path between node i and node j. Note that this bias term $b_{\phi(i,j)}$ is shared accross all layers[8].

[4]Which corresponds to the number of in-going edges.

[5]Which corresponds to the number of out-going edges.

[6]Note that for undirected graphs, $\deg^-(v_i)$ and $\deg^+(v_i)$ are unified to $\deg(v_i)$.

[7]By "global" we mean a dense all-pairs pattern: the model computes an attention score between every pair of tokens (or nodes, in our setting), rather than restricting attention to adjacent or local neighbors.

[8]Because $\phi(i,j)$ depends only on the shortest path distance between nodes i and j, which is a fixed structural property of the input graph, it does not change with depth; therefore the same bias is used in every layer.

In this way, each node can attend to all other nodes in the graph while incorporating structural information. By introducing the bias term $b_{\phi(i,j)}$, the model is able to modulate the attention score based on the shortest-path distance between nodes i and j. As a consequence, nodes that are far apart in the graph are likely to receive lower attention weights, whereas structurally closer nodes will have a stronger influence on each other.

Finally, since in many real-world graphs edges also present structural or semantic features, it becomes crucial to incorporate such information into the model[9]. For instance, in social networks edges may carry information such as the interaction type (friendship, follow, retweet), in citation networks they may encode the publication venue or the timestamp of the citation, while in knowledge graphs they are explicitly labeled with semantic relation types. Ignoring these edge-specific properties would lead to a significant loss of information, since the connectivity pattern alone does not always capture the full heterogeneity of the underlying system.

In order to integrate edge attributes into the attention computation, the Graphormer framework extends the attention mechanism by introducing an additional edge encoding term. Concretely, given an edge feature vector $\mathbf{e}_{ij} \in \mathbb{R}^{d_e}$ associated with the edge between nodes i and j, the model applies a learnable linear transformation $\mathbf{W}_e \in \mathbb{R}^{d \times d_e}$ that projects the edge features into the same latent space of the attention mechanism. The resulting projection is then combined with a learnable vector $\mathbf{u} \in \mathbb{R}^d$, producing a scalar contribution that can be seamlessly added to the attention score. The modified attention formulation becomes:

$$A_{ij} = \frac{(\mathbf{h}_i \mathbf{W}_Q)(\mathbf{h}_j \mathbf{W}_K)^\top}{\sqrt{d}} \; + \; b_{\phi(i,j)} \; + \; \mathbf{u}^\top \mathbf{W}_e \mathbf{e}_{ij}, \tag{3.1.2}$$

where:

- $\mathbf{h}_i, \mathbf{h}_j \in \mathbb{R}^d$ are the hidden representations of nodes i and j,
- $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d \times d}$ are the query and key projection matrices,
- $b_{\phi(i,j)}$ is the spatial bias term encoding the structural distance between nodes i and j in the graph,
- $\mathbf{e}_{ij}$ represents the edge feature vector,
- $\mathbf{W}_e$ and $\mathbf{u}$ are trainable parameters that allow the model to adaptively weight edge features in the attention space.

This formulation allows the attention mechanism to jointly exploit:

[9]Note that in Relbench edges do not come with specific features.

1. the global node-to-node interactions enabled by the standard Transformer formulation,

2. the structural properties of the graph topology encoded through the spatial bias $b_{\phi(i,j)}$,

3. the semantic or structural edge information carried by $\mathbf{e}_{ij}$.

From a modeling perspective, this design is particularly powerful, as it allows the network to learn fine-grained relational patterns: nodes can attend more strongly not only to structurally close nodes but also to neighbors connected by edges with highly informative features.

While the Graphormer model represents a significant step forward in adapting Transformer architectures to graph representation learning, it is important to acknowledge several intrinsic limitations that emerge from its design choices. These limitations concern both the applicability of the model and its computational feasibility.

One of the main drawbacks of Graphormer is that it has been primarily designed and evaluated on **homogeneous graphs**, where all nodes and edges belong to the same type and share a common feature space. This design choice simplifies the formulation of the spatial and centrality encodings but severely limits the direct applicability of the model to *heterogeneous graphs*, which are ubiquitous in many real-world scenarios (e.g., social networks, knowledge graphs, and relational databases). Extending Graphormer to heterogeneous settings requires non-trivial modifications to its encoding mechanisms, such as designing type-specific parameters or incorporating meta-path based reasoning, which are not covered in the original formulation.

From a computational standpoint, Graphormer inherits the well-known **quadratic complexity** of the Transformer architecture with respect to the number of nodes N. Specifically, the self-attention mechanism requires computing attention scores for every possible pair of nodes, leading to a cost of $\mathcal{O}(N^2 d)$ in time, where d denotes the hidden dimension. For large graphs, which are common in domains such as citation networks or social platforms, this quickly becomes computationally prohibitive, making Graphormer impractical without heavy sampling or graph partitioning strategies.

Another limitation lies in the spatial encoding mechanism. In order to model structural properties, Graphormer introduces learnable bias terms $b_{\phi(i,j)}$ that depend on the shortest-path distance between node pairs. While effective at encoding topology, this design results in a very large number of additional parameters, since a separate embedding must be learned for each possible distance bucket. In graphs with long average path lengths, this may lead to a dramatic growth in the parameter count, introducing both storage overhead and a risk of overfitting.

Moreover, computing and storing shortest-path distances for all node pairs is itself a costly preprocessing step.

The combination of dense global attention, degree centrality embeddings, and spatial bias terms makes Graphormer challenging to scale beyond medium-sized graphs. Unlike message-passing graph neural networks (e.g., GraphSAGE or GCN), which only propagate information locally, Graphormer enforces global connectivity at each layer. While this design captures long-range dependencies, it also leads to substantial memory overheads and slower training dynamics. As a result, Graphormer is often unsuitable for very large-scale relational datasets without careful optimization or approximation strategies.

Graphormer successfully demonstrates how Transformers can be adapted to exploit graph-specific inductive biases, but its assumptions and computational requirements restrict its applicability in heterogeneous and large-scale graph scenarios. These limitations motivate the need for lighter, more flexible extensions capable of handling relational heterogeneity and scaling to massive graphs.

HeteroTemporal Graphormer (HTG)

Building on the limitations discussed for the Graphormer[32], we start from the observation that, while powerful, it is designed for essentially homogeneous and static graphs: relation semantics are not distinguished beyond a single structural prior and time is not used in a pairwise manner. In heterogeneous, timestamped settings this leaves attention without the signals needed to decide which pairs are semantically compatible and which are temporally related, so nodes with similar features but different roles or occurring at very different moments can look deceptively alike to the model.

Our goal is therefore to tailor the architecture to heterogeneous and temporal graphs without changing what attention is, or how a Transformer operates: we keep the standard scaled dot–product, the same softmax–based weighting, and the usual projection–aggregation pipeline, and we simply expose to the logits[10] the few structural signals that truly matter in this regime. To remain computationally mindful, we introduce only lightweight terms and minor practical choices that reduce cost where possible (e.g., restricting queries to seed rows in the final layer and deliberately omitting all–pairs shortest–path distance biases within each minibatch, whose computation and storage would be prohibitive), yet we refrain from altering the mechanics of self–attention[26]. In short, the primary objective

[10]Here, for "logits" we mean the raw, pre–softmax attention scores: for head h they form a matrix $L^{(h)}$ with entries $\ell_{ij}^{(h)}$, one scalar for each pair (i, j), computed as the scaled dot product of the query of i and the key of j (plus our bias terms) before normalization.

of this architecture is to extend Graphormer (and, therefore, the Transformer architecture) to heterogeneous and temporal graphs, not to redesign attention itself.

We do so by adding a small set of trainable priors directly to the attention logits; by placing structure and time exactly where the decision is taken, these priors give the model explicit signals about node type compatibility, relation presence, and relative recency, which lets attention favour pairs that are meaningful in the graph while still being guided by feature based similarity. We call this architecture HeteroTemporal Graphormer (HTG).

Let $X \in \mathbb{R}^{N_{\text{tot}} \times C}$ be the matrix of node representations for the sampled subgraph, where each row corresponds to one node in the fused index space and each column to a feature channel, and where $C = HD$ with H heads and head size D.

Before computing attention we apply pre normalization to stabilize the scale of each row, that is $Y = \text{LN}(X)$. From Y we form three standard linear projections using learnable weight matrices W_Q, W_K, and $W_V \in \mathbb{R}^{D \times D}$, which map the input features into query, key, and value spaces: $Q = YW_Q$, $K = YW_K$, and $V = YW_V$, then we reshape them into H heads. For head h we denote by $q_i^{(h)} \in \mathbb{R}^D$ the query of row i, by $k_j^{(h)} \in \mathbb{R}^D$ the key of column j, and by $v_j^{(h)} \in \mathbb{R}^D$ the value of column j. The raw similarity for head h between node i and j is the scaled dot product

$$S_{ij}^{(h)} = \frac{\langle q_i^{(h)}, k_j^{(h)} \rangle}{\sqrt{D}},$$

which should be read as follows. The inner product $\langle q_i^{(h)}, k_j^{(h)} \rangle$ measures alignment between what node i is searching for and how node j describes itself, so larger alignment yields a larger score, and the division by $\sqrt{D}$ compensates for the fact that the inner product sums D terms, keeping scores on a comparable scale so that the softmax that follows does not saturate. Rows index the nodes that attend and columns index the candidates they may attend to, hence $S_{ij}^{(h)}$ is the score that says how much node i should consider node j under head h.

This operation is very powerful and is a central component of the transformer architecture. It learns how much one node should attend to any other node. Taken in isolation, however, it ignores the topological and temporal structure that enriches our input graph. As a result, two pairs with similar features may receive comparable scores even if they are far apart in the graph or far apart in time. By "far apart in time" we mean a large absolute lag $|\Delta t_{ij}| = |t_j - t_i|$ between the timestamp of the attending node i and that of the candidate j. In most temporal prediction problems, recent evidence (small $|\Delta t_{ij}|$) is more informative because systems drift, behavior and context change, and very old events tend to carry weaker or misleading signals. For example, when ranking which races a driver should attend to, a race from last week should typically influence the decision more than a race from three seasons ago, even if their aggregate features look similar.

Similarly, by "structurally distant" we mean that the shortest path distance $d_G(i, j)$ in the sampled graph is large, that is, many edges must be traversed for information to flow from i to j. Long chains dilute signal and increase the chance that attention is driven by confounding paths rather than by direct, actionable relations. For example, for a seed *driver* node, the most reliable signals come from its own recent *results* entries (1 hop) or from the *constructor* it raced for in those events (2 hops: *driver* $\rightarrow$ *results* $\rightarrow$ *constructor*). By contrast, evidence routed through a long chain such as *driver* $\rightarrow$ *results* $\rightarrow$ *race* $\rightarrow$ *circuit* $\rightarrow$ *other race* $\rightarrow$ *results* $\rightarrow$ *other driver* spans many hops and mixes contexts across circuits and seasons; even if the distant driver's features look similar, this path is more likely to be confounded than informative.

We therefore adopt this operation and introduce learnable bias terms that account for structural, relational, and temporal factors. In particular, we add three priors in logit space as bias terms. Doing so we preserve the usual dot product geometry and the core computation, while the biases act as small priors that gently shift attention toward pairs suggested as plausible by graph semantics and chronology, yet still let feature based similarity dominate when it is strongly informative. This design offers simple and controllable access to node type compatibility, relation presence, and temporal proximity with a minimal number of extra parameters and without changing the self-attention mechanism.

First, we introduce a *type–pair prior* to make explicit, at the level of node types, who should talk to whom. In heterogeneous graphs, two nodes can look similar in feature space yet play very different roles; conversely, nodes with distinct features may routinely interact because their types are complementary. Relying on the model to infer these regularities only through learned projections of the features can be data–hungry and brittle: it forces attention to "rediscover" global type compatibilities from many local examples, and small distribution shifts can erase that signal. The type–pair prior injects this knowledge directly where the decision is taken, as a small, learnable log–offset that biases attention toward type combinations that are known to be meaningful in the domain (e.g., Driver$\rightarrow$Race) and away from those that are usually irrelevant. It is *asymmetric* by construction, so it can express that $A \rightarrow B$ is common while $B \rightarrow A$ is not, and it is *head–specific*, allowing different heads to specialise on different interaction patterns without competing within the same parameters. Importantly, this prior does not alter the mechanics of attention (queries, keys, and values are computed exactly as usual) yet it improves sample efficiency and stability by providing a low–dimensional, interpretable bias that resolves feature–level ambiguities early.

Let T be the number of node types and let $\tau : \{1, \ldots, N_{\text{tot}}\} \rightarrow \{1, \ldots, T\}$ map each fused index to its node type; the parameterization that follows encodes a compatibility score for each ordered (source type, destination type, head) triple and is learned end–to–end from data while being initialized to zero so as not to

dominate when features are already decisive.

We learn a tensor $B_{\text{type}} \in \mathbb{R}^{T\times T\times H}$ and define, for head h:

$$b^{(h)}_{\text{type}}(i,j) = B_{\text{type}}\Big[\tau(i),\, \tau(j),\, h\Big].$$

This scalar is added to the logit $\ell^{(h)}_{ij}$, hence

$$\ell^{(h)}_{ij} \leftarrow \ell^{(h)}_{ij} + b^{(h)}_{\text{type}}(i,j).$$

Operationally, the calculation is a table lookup indexed by the source type of row i and the destination type of column j for each head. In vectorized form we gather B_{type} with the type vectors of rows and columns to obtain a dense bias matrix per head, so the cost is negligible. Intuitively, this bias term acts as a small prior that raises or lowers the baseline score before the softmax according to type compatibility. The parameters are learned end to end and are initialized to zero, so when features are strongly informative the scaled dot product dominates, while in ambiguous cases the prior gently nudges attention toward type compatible pairs.

Second, we add a relation adjacency prior that makes the model aware of which pairs are actually connected and by which relation. Let R be the number of relation types and, for each $r \in \{1,\dots,R\}$, let $E_r \subseteq \{1,\dots,N_{\text{tot}}\}^2$ be the set of directed edges of relation r. We learn one scalar per relation and per head, collected in $B_{\text{rel}} \in \mathbb{R}^{R\times H}$, and we add to the logit of head h the term

$$\Big(b^{(h)}_{\text{rel}}\Big)_{ij} = \sum_{r=1}^{R} \mathbf{1}\Big\{(i,j) \in E_r\Big\}\, B_{\text{rel}}[r,h].$$

This is a simple computation with a clear meaning. If there is an edge from i to j of type r, the score $\ell^{(h)}_{ij}$ receives the constant offset $B_{\text{rel}}[r,h]$ before the softmax, otherwise it receives nothing. In matrix form this is the linear combination

$$B^{(h)}_{\text{rel}} = \sum_{r=1}^{R} B_{\text{rel}}[r,h]\; A_r, \qquad (A_r)_{ij} = \mathbf{1}\Big\{(i,j) \in E_r\Big\}.$$

The benefit is twofold. First, the prior injects local topology in the most direct way, it tells attention that immediate neighbours are special and that their importance can differ across relation types and across heads. Second, it is parameter efficient and stable, since it only adds one number per relation and head and it leaves the dot product geometry untouched, so when features are strongly informative the dot product still dominates. The parameters are learned end to end and are initialized to zero, which means the model discovers on its own whether a relation should amplify or dampen attention for a given head.

A short example clarifies the effect. Suppose nodes of type Driver connect to nodes of type Race with two relation types, participates_in and won. If driver

i has edges $(i, j_1) \in E_{\text{participates\_in}}$ and $(i, j_2) \in E_{\text{won}}$, then for head h the logits receive

$$\ell_{ij_1}^{(h)} \leftarrow \ell_{ij_1}^{(h)} + B_{\text{rel}}[\text{participates\_in}, h], \qquad \ell_{ij_2}^{(h)} \leftarrow \ell_{ij_2}^{(h)} + B_{\text{rel}}[\text{won}, h].$$

Intuitively, if the objective places greater value on victories than on mere participation, the learned offsets can reflect this preference by assigning a larger value to $B_{\text{rel}}[\text{won}, h]$ than to $B_{\text{rel}}[\text{participates\_in}, h]$ for some heads. In that case, all else being equal, edges labelled *won* contribute a larger additive term to the logit and thus tend to attract more attention. Nothing is hard–coded here: the relation offsets are initialized near zero and adjusted by the data and the training objective, so the model adopts such a preference only insofar as it improves the task loss. If a pair (i, j) has no edge, its logit receives no relation offset and remains governed by the scaled dot product and by the other priors. In practice the term is applied sparsely, only at the pairs that appear in the edge lists E_r, so it is cheap to compute and it works in concert with the type pair prior and the temporal prior to guide attention toward the graph structure that matters.

Finally, we add a seed wise temporal prior that teaches attention to use time in a direct and pairwise way. In temporal graphs, absolute timestamps inside node features are often not enough, because the decision of who to attend to depends on the relative time between the seed node we supervise and the candidate node (or neighbour) it may look at. We therefore add to the logits a term that depends on the time difference from the seed row to each neighbour node. Let S be the set of seed nodes in the current minibatch, let t_i denote the timestamp of node i, and define the directed lag $\Delta t_{ij} = t_i - t_j$, which is always positive[11]. For each seed node i and candidate neighbour j, we define the nonnegative lag

$$d_{ij} := t_i - t_j \quad \geq 0,$$

which measures how far in the *past* node j is with respect to the attending seed node i. We do not use d_{ij} directly because its dynamic range can span orders of magnitude and the raw scale is sensitive to the choice of time unit and to outliers; using it as is would either force the model to learn a brittle internal rescaling or make optimization unstable. Instead, we discretize d_{ij} so that very recent events are resolved finely and very old events are compressed. To turn the continuous,

[11]By construction of our temporal neighbor sampling, candidate nodes j in each minibatch are never in the future of the attending seed i (i.e., $t_j \leq t_i$). This prevents data leakage and implies $(t_i - t_j) \geq 0$ for all pairs considered. In practice we filter out neighbors with $t_j > t_i$, so the temporal prior only needs to parameterize nonnegative lags (with $(t_i - t_j) = 0$ allowed for same–time events). *Equivalently*, if one defines the lag as $\Delta t_{ij} = t_j - t_i$, then $\Delta t_{ij} \leq 0$ under the same sampling rule.

nonnegative lag $d_{ij} = t_i - t_j$ into a small number of discrete categories, we *bucketize* it into K intervals. Here $K \in \mathbb{N}$ is an explicit hyperparameter that sets the temporal resolution of the prior: we partition the (log–compressed) lag axis into K contiguous bins and assign all lags falling in the same bin the *same* learned offset. Formally, this yields an index $\text{bucket}(d_{ij}) \in \{0, \ldots, K-1\}$ that selects one of K levels in a head–wise table $E_{\text{temp}} \in \mathbb{R}^{K \times H}$. A larger K provides finer granularity, while a smaller K gives a more compact parameterization. A bucket is a closed interval of lag values, and all lags falling in the same interval share the same learned offset; this replaces an unbounded continuous input with a few learnable levels, improving statistical robustness and keeping the computation stable.

Before discretization we apply a *logarithmic compression*, concretely $u(d) = \log(1+d)$, so that very recent lags are resolved finely while increasingly distant lags are progressively compressed. The reason for the logarithm is twofold. First, it makes the encoding *relative* rather than purely absolute: the increment $u(d+\delta) - u(d) \approx \delta/(1+d)$ shows that the same absolute change δ produces a large effect when d is small (where recency matters) and a vanishing effect when d is large (where all very old events are similarly uninformative). This illustrates that recency matters and that very old events are almost equally uninformative. Second, it greatly reduces sensitivity to the choice of units and to heavy-tailed noise: changing from days to hours multiplies d by a constant, which becomes an *additive* shift in $u(d)$ that is later absorbed by normalization and capping, whereas without the log the whole scale of the prior would be distorted.

We also introduce a *temporal horizon* $U > 0$, which caps the effective range of lags: any $d_{ij} > U$ is treated as "equally old" and mapped to the last bin. This prevents rare, very large gaps from stretching the scale and ensures that modeling capacity is concentrated near $d_{ij} = 0$. Operationally, we compress lags with $\log(1+d_{ij})$, clip them at $\log(1+U)$, and then normalize by the same quantity so that $d_{ij} = 0$ maps to 0 and all $d_{ij} \geq U$ map to 1 before quantization into the K bins.

The logarithm scale, plus the (U, K) discretization yields a simple, monotone, unit-stable representation of recency that allocates modeling capacity where it is most useful and avoids wasting parameters on the long, uninformative tail of very old events.

Let $U > 0$ be a cap (temporal horizon) and let K be the number of buckets; we map

$$\text{bucket}(d_{ij}) \;=\; \left\lfloor \frac{\min\Big\{ \log\Big(1 + d_{ij}\Big),\; \log(1+U) \Big\}}{\log(1+U)} \,(K-1) \right\rfloor \;\in\; \{0, \ldots, K-1\}.$$

Where, $\log(1+d_{ij})$ is a monotone compression of the nonnegative lag d_{ij}: it preserves order, gives fine resolution near 0 where recency matters, and progressively

compresses large lags so that very old events do not dominate. The $\min\{\cdot, \log(1+U)\}$ applies a hard cap at the *temporal horizon* U: all lags beyond U are treated as "equally old", which prevents rare outliers from creating spurious extra levels and stabilizes learning. The division by $\log(1+U)$ normalizes the compressed value to [0,1], making the construction scale-free with respect to the chosen horizon: 0 maps to 0 and any $d_{ij} \geq U$ maps to 1. Multiplying by $(K-1)$ then rescales [0,1] to the continuous interval $[0, K-1]$, so that we have room for exactly K discrete indices (from 0 to $K-1$). Finally, the floor $\lfloor \cdot \rfloor$ selects the integer index of the bin: $d_{ij} = 0$ yields bucket 0, any $d_{ij} \geq U$ yields bucket $K-1$, and intermediate values are assigned to contiguous bins that grow wider as d_{ij} increases (because the partition is uniform in $\log(1+d)$, not in d). Equivalently, the bin boundaries in the original time scale are

$$\tau_m \;=\; (1+U)^{\,m/(K-1)} - 1, \qquad m = 0,1,\ldots,K-1,$$

so the intervals are $[\tau_m, \tau_{m+1})$ and expand geometrically, which is exactly the desired behaviour for a recency prior: many narrow buckets near 0 and few wide buckets in the far past. This sequence of capping, normalizing, rescaling, and flooring thus converts a potentially heavy-tailed, unit-sensitive lag into a small, ordered set of stable categories that the model can parametrise with one learned offset per bucket.

The bucket index is then converted into a head-specific logit offset via a learned table $E_{\text{temp}} \in \mathbb{R}^{K \times H}$:

$$b^{(h)}_{\text{temp}}(i,j) \;=\; E_{\text{temp}}\Big[\text{bucket}(d_{ij}),\, h\Big], \qquad \ell^{(h)}_{ij} \leftarrow \ell^{(h)}_{ij} + b^{(h)}_{\text{temp}}(i,j).$$

In words, recent pairs ($d_{ij} \approx 0$) fall into small-index buckets that the model can learn to reward, while distant past events are pushed into outer buckets with coarser resolution. This makes *recency* an explicit, learnable signal at the very place where the attention decision is taken, without changing the standard Q–K–V computation.

The prior is head specific, hence different heads can specialise, for example one head can focus on very recent evidence, another on medium lags, another on long term context.

For completeness, we also attempted to add a *dedicated spatial bias term* on top of our relation and temporal priors, in the spirit of Graphormer's structural encoding: the idea was to inject into the logits an additional bias term derived from the shortest–path distance $d_G(i,j)$ between nodes i and j (e.g., clipping d_G to a radius and mapping it to an embedding $E_{\text{spd}}[\text{clip}(d_G(i,j))]$). The motivation is intuitive: small graph distance should gently increase the score, while very distant or disconnected pairs should not be favored. However, in our heterogeneous, time–stamped, *sampled* regime it proved impractical and of limited benefit. Because each minibatch is a fresh subgraph produced by neighbor sampling, all–pairs

distances cannot be precomputed and reused; one must either recompute many BFS/SSSP runs on the fly for the batch graph (time–intensive), or materialize a dense $N \times N$ distance map per batch (memory–intensive), both of which scale poorly as node counts grow. Heterogeneity and directionality make things worse: distances are undefined for many ordered pairs in directed, disconnected subgraphs, requiring ad–hoc fallbacks that blur semantics, further increasing complexity. Despite the considerable computational and implementation complexity, the observed improvements in our tests were negligible, often within run-to-run noise, and in several settings statistically indistinguishable from zero. We therefore removed the shortest–path bias and retained only relation and time priors.

Putting everything together, the attention logits for head h are

$$\ell_{ij}^{(h)} = \frac{\langle q_i^{(h)},\, k_j^{(h)} \rangle}{\sqrt{D}} + B_{\text{type}}[t_i, t_j, h] + \sum_{r=1}^{R} \mathbf{1}\{(i,j) \in E_r\}\, B_{\text{rel}}[r, h] + \left(B_{\text{temp}}^{(h)}\right)_{ij},$$

that is, the usual similarity plus three interpretable log priors. Weights are obtained by a row wise softmax

$$\alpha_{ij}^{(h)} = \frac{\exp\left(\ell_{ij}^{(h)}\right)}{\sum_{u=1}^{N_{\text{tot}}} \exp\left(\ell_{iu}^{(h)}\right)},$$

which turns logits into non negative coefficients that sum to one on each row, hence each row forms a probability distribution over columns. The head output is a weighted average of values

$$z_i^{(h)} = \sum_{j=1}^{N_{\text{tot}}} \alpha_{ij}^{(h)}\, v_j^{(h)},$$

and the multi head output is $z_i = \left[z_i^{(1)} \parallel \ \ldots \ \parallel z_i^{(H)}\right] W_O$,[12] followed by dropout and a residual connection with X, then a position wise feed forward in pre norm. Intuitively this computation says the model first decides who should talk to whom via an alignment between queries and keys, then it gently biases that decision with three small learned terms that encode which types usually interact, whether an actual edge of a given relation exists, and how recent a node is with respect to a seed, and finally it aggregates the content of the chosen neighbours through the values.

3.2 Pre-training strategies

Pre-training, particularly in its self-supervised form, has been widely adopted across multiple fields. In natural language processing, it has matured into a

[12]Where W^{O} is the output projection matrix that maps the concatenated attention heads back to the model dimension.

scalable paradigm for learning general-purpose representations from unlabeled data, achieving state-of-the-art results. Beyond NLP, pre-training has also been successfully applied to image classification [54], speech recognition [55], multimodal vision–language [56], and many others.

In contrast, for graph data, especially in the *heterogeneous* and *temporal-aware* settings, we still lack a comparable, scalable, and consistently effective paradigm, with objectives and benchmarks that translate into reliable downstream gains [57, 58].

Self supervised pretraining leverages abundant unlabeled interactions to learn type aware and time aware representations that capture relational structure and temporal regularities. By training on large historical prefixes of the graph without manual labels, the encoder integrates information across heterogeneous entities and relations while remaining grounded in time consistent neighborhoods. The resulting embeddings provide a reusable substrate for diverse downstream tasks, reduce the need for task specific supervision, and improve robustness to changes in the data over time.

Pre-training on graphs is motivated by two persistent challenges. First, standard GNNs exhibit poor *out-of-distribution* (OOD) robustness: performance degrades when topology, attributes, scale, or even label mechanisms shift. These phenomena are very common in graphs due to non-i.i.d. dependencies, long-range interactions, and (in heterogeneous settings) type shifts (so performance degrades under distribution shifts between training and test graphs). Second, labeled data are scarce and costly.

Pre-training addresses this gap by distilling structure-, type-aware priors from large unlabeled signals, yielding representations that transfer more robustly under distribution shift and reduce label dependence.

Graph pre-training helps the model building a principled understanding of the data: it internalizes the graph's topological structure (who connects to whom and at which scales), the types of relations (edge semantics and interaction patterns), and the semantic content of entities and links (node/edge attributes and their meaning). Consequently, the resulting representations are less brittle under *distribution shift*, changes in connectivity, feature marginals, scale and transfer more reliably to new nodes, relations, or future time steps [59]. By pre-exposing the model to unlabeled graphs, pre-training teaches it the common patterns of topology, relation types, and attribute context before any task-specific supervision. This lowers the number of labeled examples required, reduces sample complexity and improves performance when labels are scarce. It also stabilizes optimization: a pre-trained initialization yields better-conditioned losses, faster and more reliable fine-tuning on small or imbalanced graph datasets.

We developed several *self-supervised* pre-training schemes that aim to leverage the raw graph signals. Our pre-training strategies are conducted on the same target

graph (in-graph pre-training), using its structural and temporal signals as self supervision, before fine-tuning on a small labeled split of that same graph, while enforcing strict anti-leakage (e.g., temporal splits). Thus, our approach enriches topological and semantic representations in the GNN, enabling robust learning, even when the label signal is too weak for the task's complexity.

Prior works[59, 60] on pretraining on graphs has largely emphasized transfer across datasets, pretraining on a source graph (or many source graphs) and then adapting to different target graphs. In this work, instead, we perform in domain self supervised pretraining restricted to the training split of the same dataset used for the downstream task; no external graphs or labels are used. This choice is motivated by three factors. First, there is substantial schema and semantic mismatch across graph databases, including node and edge types, relation semantics, metapaths, and keys, which reduces the portability of type aware and structure aware priors. Second, there are pronounced covariate and temporal misalignments across domains, since node and edge features often differ in range, sparsity, correlations, and meaning, and dynamic graphs are sampled at incompatible temporal granularities; As a result, temporal encoders, positional schemes, and normalizations learned on a source domain become miscalibrated on the target, which increases the risk of negative transfer. Third, there are engineering constraints such as schema alignment, feature unification, and multi domain SSL objectives and samplers that fall outside the scope of this work.

By learning priors within the same schema, feature space, and timescale as the downstream task, in domain pre-training produces representations that are better matched to the target distribution and therefore more label efficient and more robust under realistic shifts within the domain.

Our focus is therefore to measure the true benefit of in-domain pre-training while avoiding confounders due to domain shift. All statistics (normalizations, vocabularies) are estimated on the training split only. We leave the exploration of cross-dataset pre-training and domain-bridging strategies to future work.

3.2.1 Masked Attribute Prediction (MAP) Pretraining

Building on masked graph autoencoder methods[61, 62, 63, 64, 65], we adopt masked attribute prediction in our in domain, time aware, heterogeneous setting, and implement it with batch only masking, sentinel tokens, and column specific lightweight heads, while keeping the objective restricted to masked positions.

We aim to train the model to reconstruct each masked attribute using the graph's topology and the node's remaining attributes, encouraging it to capture and internalize the semantic relations encoded in the network.

In each time-consistent mini-batch[13] drawn from the training split, we first choose a subset of node types and columns, then mask some of their attribute values. The masking process should cover both categorical and numerical fields, while leaving all other features and the graph topology untouched. The model is then optimized to reconstruct the masked entries from the remaining features and the surrounding relational context, with the objective computed exclusively on masked positions.

Crucially, masking is performed batch-only: we never alter the global database object, nor normalization statistics, thereby avoiding permanent edits, precluding cross-iteration leakage, and preserving the in-domain data distribution throughout training.

The masking policy is **task aware**. We do not sample mask positions uniformly across the schema. Although arbitrary sampling would be more onerous in practice, increasing memory usage and requiring additional preprocessing to maintain per column vocabularies and normalizers, the central issue is statistical: many columns are highly noisy, rare, and inconsistently encoded, with substantial missingness and small effective sample sizes; as a result, distributional estimates and normalizers become unstable, gradients exhibit high variance, and the model tends to fit idiosyncratic noise rather than meaningful structure, which degrades generalization and lowers the signal to noise ratio.

Therefore, we first identify the tables and attributes that carry signal for the downstream objective and we apply the masking process only to them. In particular, at the table level, we run **ablation studies** by retraining the same model while excluding one table at a time, then we rank tables by the induced drop in the validation metric. Within each table, we repeat the analysis at the column level by excluding groups of columns and measuring their marginal effect on the validation score. We then restrict the maskable pool to attributes whose ablation consistently degrades validation performance, that is, attributes whose removal produces a reproducible drop in the validation metric across seeds and splits. In addition to this ablation based selection, we apply a structural relevance check: we exclude attributes that are weakly dependent on graph structure or local context, such as raw identifiers or free form names that cannot be inferred from neighbors or from other features.

We hope that this careful selection concentrates the pre training signal on informative variables, produces stronger gradients during reconstruction, and reduces the risk of a degenerate objective in which the masked value is either unpredictable or trivially recoverable.

[13]By *time consistent mini batch* we mean that, for each seed node at time t_v, we construct a time consistent subgraph that includes only its neighbors with timestamps $\leq t_v$.

As a concrete example on the rel trial database, we restrict the maskable attributes to variables that demonstrably contribute to the downstream objective, as identified through table level and column level ablation on this dataset. In the studies table this set includes target duration, study type, phase, enrollment, and the FDA regulation indicator. In the outcomes table it includes outcome type and parameter type. The same systematic ablation procedure is applied table by table for every database we use, and each dataset derives its maskable pool from its own table level and column level analyses.

Importantly, during pretraining, we alternate which attributes are hidden across iterations; this limits overfitting to any single feature and encourages the encoder to rely on relational structure and contextual signals. In practice, at each pretraining epoch, the temporal neighbor sampler produces a time consistent subgraph. We then restrict masking to the columns and tables designated by the ablation studies and, for each of these columns, we sample an independent Bernoulli mask over rows; we apply it to replace the selected entries in the current batch with a sentinel value, and we store the corresponding ground truth values together with the row indices of the batch.

Given the masked mini-batch, we pass the data through the encoder pipeline in three stages. First, the feature encoders map raw attributes to dense vectors while applying the mask. For masked positions the true values are withheld and replaced with a type specific sentinel code, that is a dedicated token id for categorical fields and a reserved numeric code for continuous fields. Encoders treat the sentinel strictly as an unknown marker. The withheld ground truth is stored separately and is used only to compute the reconstruction loss. Second, the temporal module enriches these representations with temporal context[14]. Third, a heterogeneous GNN performs message passing over the typed edges to integrate relational context across neighboring nodes. This sequence produces a node-level embedding z_i for every node i in the batch, which compactly summarizes attribute, temporal, and structural signals and is then used by the decoder to reconstruct the masked attributes.

The **decoder** attaches a simple head for each maskable attribute, defined at the level of (*node type*, *column*)[15]. For categorical attributes we use a linear layer $\mathbb{R}^d \to \mathbb{R}^{|\mathcal{C}|}$, where $|\mathcal{C}|$ is the number of unique classes computed once on the training database and then kept fixed; during the reconstruction loss the output is interpreted with a softmax. For numerical attributes we use a linear layer $\mathbb{R}^d \to \mathbb{R}$ that produces a scalar prediction, which we compare with the ground truth using a regression loss such as mean squared error. Heads are instantiated independently for

[14]Implemented with the HeteroTemporalEncoder described earlier.

[15]In practice, each masked column in each table uses its own decoder head.

each pair (*node type*, *column*), and there is no parameter sharing across attributes or types. We keep these heads intentionally simple to concentrate modeling capacity in the encoder GNN, which is the component reused in downstream tasks; this separation encourages the encoder to internalize structure, type information, and context, while the heads simply map the shared representation to attribute specific outputs.

At pre-training time, only the heads corresponding to columns that are actually masked in the current mini batch are active. Each active head takes the node embedding z_i and produces predictions only for the rows indicated by the mask indices that are local to the batch. For categorical attributes, the head outputs class logits and is optimized with cross entropy on the masked entries; for numerical attributes, the head outputs a scalar and is optimized with mean squared error on the masked entries. The reconstruction loss is computed exclusively on masked positions, summed across all active heads in the batch, and backpropagated through both the decoder heads and the encoder. This objective encourages the encoder to combine residual feature evidence with relational and temporal context in order to recover the missing entries, while the heads act as attribute specific readouts.

This design targets the encoder: by optimizing attribute specific heads on a shared d-dimensional node representation, the pretraining objective drives the encoder to integrate graph topology, relation types, and contextual semantics into a transferable embedding. The heads act only as attribute specific readouts, while the encoder begins to exploit graph structure and accumulates graph aware features that are reused in downstream tasks. This pretraining strategy is modular and scalable: the number of additional parameters grows linearly with the count of maskable attributes, and the simplicity of the heads reduces the risk that decoders overfit to idiosyncrasies of any single table or column.

For each node type nt, let $\mathcal{S}_{nt}$ denote the predefined maskable set. At every training step, given the sampled mini-batch, we consider only those columns $c \in \mathcal{S}_{nt}$ that are present in the batch table for nt. For each such pair (nt, c), we draw an independent Bernoulli mask with fixed rate $p \in (0,1)$ over the batch rows of that table and hide the selected entries, while caching the corresponding ground-truth values and their batch-local indices. The rate p is a hyperparameter (we set $p = 0.3$), and the mask is resampled at every step, so the locations of the hidden entries vary across iterations. By rotating which attributes are masked at each pretraining epoch, the model is prevented from relying on any single feature and is encouraged to exploit relational and temporal context that transfers more reliably to the downstream objective.

Formally, letting $\mathcal{M}$ be the set of masked column instances in the current batch, with items identified by the node type t, the column name c and the masked row

indices $I_{t,c}$, the per-batch objective is

$$\mathcal{L}_{\text{MAP}} = \sum_{(t,c)\in\mathcal{M}} \begin{cases} \sum_{i\in I_{t,c}} \text{CE}\Big(g_{t,c}(z_i^{(t)}),\, y_i^{(t,c)} \Big) & \text{if } c \text{ is categorical,} \\ \sum_{i\in I_{t,c}} \Big\| g_{t,c}(z_i^{(t)}) - y_i^{(t,c)} \Big\|_2^2 & \text{if } c \text{ is numerical,} \end{cases}$$

where $g_{t,c}$ is the linear head attached to column c of node type t, $z_i^{(t)}$ is the encoder embedding for the i-th node of type t in the *batch* and $y_i^{(t,c)}$ is the ground-truth attribute value saved before masking.

We now report simplified pseudo-code aligned with our actual code of this pre-training strategy:

Algorithm 2 MAP-Pretrain (batch-only masking)

procedure MapPretrain($\mathcal{L}$, p_{mask}, θ, η)
  ▷ $\mathcal{L}$: mini-batch loader; p_{mask}: Bernoulli rate; θ: params; η: optimizer
  for all batch $\in \mathcal{L}$ do
    (batch*, GT, Mask) $\leftarrow$ MaskAttributes(batch, p_{mask})
    $Z_0 \leftarrow$ HeteroEncoder(batch*)
    $Z_t \leftarrow$ HeteroTemporalEncode(Z_0, batch*)
    $E \leftarrow$ Edges(batch*)
    $H \leftarrow$ HeteroGNN(Z_t, E)
    $\hat{Y} \leftarrow$ MapDecode(H)
    $\mathcal{L}_{\text{cat}} \leftarrow \text{CE}\big(\hat{Y}_{\text{cat}}[\text{Mask}_{\text{cat}}],\, \text{GT}_{\text{cat}}\big)$
    $\mathcal{L}_{\text{num}} \leftarrow \text{MSE}\big(\hat{Y}_{\text{num}}[\text{Mask}_{\text{num}}],\, \text{GT}_{\text{num}}\big)$
    $\mathcal{L}_{\text{batch}} \leftarrow \mathcal{L}_{\text{cat}} + \mathcal{L}_{\text{num}}$
    $\theta \leftarrow \text{Update}\big(\theta,\, \nabla_\theta \mathcal{L}_{\text{batch}},\, \eta\big)$
  end **for**
end procedure

At each step, a mini-batch is drawn from the training split and passed to `MaskAttributes`. This routine samples an independent Bernoulli mask over the batch rows for each selected (*node type*, *column*), caches the corresponding ground-truth values together with their batch-local indices, and overwrites the entries in the batch tables with a sentinel.

The masked batch is then processed by the `HeteroEncoder`, which maps raw attributes to dense vectors while treating sentinels as unknowns rather than usable signals. Next, the `HeteroTemporalEncoder` augments these vectors with time information. The resulting node features are fed to the heterogeneous GNN `HeteroGNN`, as shown above, which yields per-node embeddings z_i.

On top of these embeddings, `MapDecode` instantiates a lightweight, column-specific decoder head for each masked attribute (*node type*, *column*). The heads share architecture but not parameters, and only those corresponding to columns actually masked in the current batch are activated. Each active head produces predictions exclusively at the masked indices: for categorical attributes it outputs class logits, whereas for numerical attributes it produces a scalar. The decoder is a small module dictionary indexed by `"{node_type}__{col}"` with a unified `forward` that iterates over the masked fields present in the current batch, gathers the corresponding node embeddings and accumulates the appropriate loss.

Training uses losses restricted to masked positions. `CE` denotes the cross-entropy computed on the masked categorical entries, and `MSE` denotes the mean squared error on the masked numerical entries. These terms are summed to form the batch objective, which is backpropagated through both the decoder heads and the encoder stack. Finally, `Update` applies the optimizer step to the model parameters. Because masking is resampled at every iteration and never written to the global database, no cross-iteration leakage can occur and the model is compelled to use residual features, temporal cues, and relational structure to reconstruct the hidden values.

The pretext objective is **deliberately asymmetric**: the decoder is kept intentionally simple (a small, column-specific head applied only at masked indices) so that most of the modeling burden falls on the encoder. Because the ground-truth value of a masked attribute is removed from the input, the model cannot recover it by copying or memorizing per-row identities. Success is only possible by exploiting structure available elsewhere: statistical regularities among the remaining attributes of the same node (co-occurrence patterns and functional constraints), relational signals that propagate through the heterogeneous graph (information from typed neighbors and their attributes), and temporal cues injected by the time encoder (causal ordering and short-term dynamics). The loss is computed exclusively at masked positions, which prevents the decoder from being rewarded for trivial predictions on unmasked entries and forces useful information to flow back through the encoder. As masks are resampled at every step, the task covers many missingness patterns and consistently pressures the encoder to build denoised, context-aware representations that generalize to downstream objectives. Masking inside the batch guarantees that we supervise exactly and only those nodes for which embeddings were computed in the current forward pass, and that the supervision signal is derived from the training split without any contamination from the future. Alternating which attributes are hidden across iterations prevents the trivial solution of memorizing a single column; conversely, avoiding masking *all* columns of a node at once ensures a residual signal remains available for meaningful reconstruction. After a few epochs the encoder has learned to predict missing attributes from context; we then discard the MAP heads and reuse the pretrained encoder weights for the downstream task.

3.2.2 Variational Graph Autoencoding Pretraining

In this pretraining strategy we adopted an in-domain, self-supervised approach based on a variational graph autoencoder[66] trained directly on the time-consistent subgraphs produced by the temporal neighbor sampler[4].

The variational graph autoencoder proposed by Kipf et al.[66] was designed to work for homogeneous, undirected and static graphs[16]. The idea we propose here is to adapt this framework for relational and temporal graphs.

Before delving into this model design is important to understand all the key pillar on which this work is built on.

A Variational Graph Autoencoder (VGAE) takes inspiration from the **autoencoder**, which is a neural architecture trained to reconstruct its inputs: an encoder maps x to a lower-dimensional code $h = f_\phi(x)$ and a decoder maps back h to the initial input space: $\hat{x} = g_\theta(h)$. The parameters (ϕ, θ) are learned by minimizing a **reconstruction loss**. The reconstruction error is usually measured with a mean squared error (MSE) between each input $x_i \in \mathbb{R}^d$ and its reconstruction $\hat{x}_i = g_\theta(f_\phi(x_i))$:

$$\mathcal{L}_{\text{rec}}(\phi, \theta) = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{d} \left\| x_i - \hat{x}_i \right\|_2^2.$$

where N is the number of training examples, d is the feature dimension of each example, x_i denotes the i-th input vector, $\hat{x}_i$ its reconstruction, and $\| \cdot \|_2$ is the Euclidean norm. The factor $1/d$ is an optional normalization by feature dimension.

Minimizing $\mathcal{L}_{\text{rec}}$ encourages the code $h = f_\phi(x)$ to capture the salient structure of the data.

[16] By static, we mean that no temporal information about the entities is available.

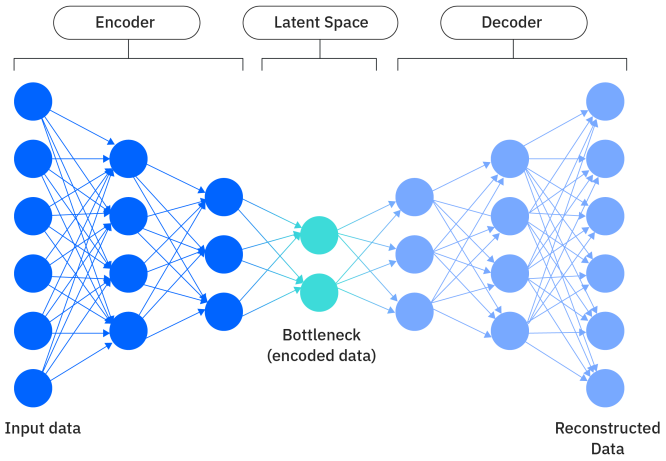


Figure 3.1: Graphical representation of an Autoencoder neural network. It's composed of an encoder sub-module, which compresses the input dimensionality to mantain only the most essential informations of the input feature space; and a decoder sub-module, which tries to de-compress the input data as close as possible to the original one. Image taken from[67].

A **Variational Autoencoder** (VAE) is a probabilistic **generative** model that learns a latent variable representation of the data. It places a simple prior over latents, typically a standard Gaussian $p(z) = \mathcal{N}(0, I)$, and specifies a decoder that defines the likelihood $p_\theta(x \mid z)$[17]. Because exact Bayesian inference of $p_\theta(z \mid x)$ is not tractable with neural decoders, the model uses an *amortized* Gaussian approximation

$$q_\phi(z \mid x) = \mathcal{N}\Big(\mu_\phi(x),\, \mathrm{diag}(\sigma^2_\phi(x))\Big),$$

with an encoder that outputs the mean and (diagonal) standard deviation of the latent code for each x.

Rather than deterministically mapping x to a single code, the VAE models a *distribution* over codes and trains by maximizing the evidence lower bound (ELBO):

$$\mathcal{L}_{\mathrm{ELBO}}(x) = \mathbb{E}_{q_\phi(z|x)}\Big[\log p_\theta(x \mid z)\Big] - \mathrm{KL}(q_\phi(z \mid x) \,\|\, \mathcal{N}(0, I))\,.$$

Where:

- x denotes the observed data. In VGAE this often means $x = (X, A)$, where $X \in \mathbb{R}^{N\times d}$ are node features and $A \in \{0,1\}^{N\times N}$ is the adjacency.

[17]Operationally, given a latent sample z, the decoder outputs the parameters of $p_\theta(x \mid z)$ (e.g., mean or logits). A reconstruction $\tilde{x}$ can be obtained by sampling from this distribution, or by taking its mean, while training maximizes $\log p_\theta(x \mid z)$ rather than requiring an explicit sample.

- z are the latent variables (e.g., per-node embeddings $Z = \{z_i\}_{i=1}^{N}$, $z_i \in \mathbb{R}^{d_z}$).
- $q_\phi(z \mid x)$ is the encoder's *approximate posterior*, typically Gaussian with diagonal covariance:

$$q_\phi(z \mid x) = \mathcal{N}\Big(z;\, \boldsymbol{\mu}_\phi(x), \mathrm{diag}(\boldsymbol{\sigma}_\phi^2(x))\Big).$$

- $p_\theta(x \mid z)$ is the decoder *likelihood* of the observed data given the latents (its form depends on what we reconstruct).
- $p(z)$ is the prior over latents, usually factorized standard normal $p(z) = \prod_i \mathcal{N}(0, I)$.
- $\mathrm{KL}(\cdot\|\cdot)$ is the Kullback–Leibler divergence; the expectation $\mathbb{E}_{q_\phi}$ is estimated via Monte Carlo using the reparameterization above.

In practice, the first term encourages accurate reconstruction of x from latent samples z (similar to the original Autoencoder), while the KL term regularizes the encoder so that its posteriors remain close to the simple prior, yielding a smooth, well-covered latent space.

To enable end-to-end learning with gradients, point are sampled via the reparameterization trick, $z = \mu_\phi(x) + \sigma_\phi(x) \cdot \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, I)$.

The main difference with a standard autoencoder approach lies in the output of the encoder module: in the autoencoder version this is a deterministic compressed representation of the input, while in the variational version the encoder returns two outputs that parameterize a diagonal Gaussian, namely (i) a mean $\boldsymbol{\mu}(x)$ and (ii) a log-variance $\log \boldsymbol{\sigma}^2(x)$.

A **Variational Graph Autoencoder** (VGAE) is a variational autoencoder applied to graphs: given a graph with adjacency A (and optional node features X), the encoder outputs, for each node i, the parameters $(\boldsymbol{\mu}_i, \log \boldsymbol{\sigma}_i^2)$ of a diagonal Gaussian $q_\phi(z_i \mid A, X)$. Latents are sampled with the reparameterization trick $z_i = \boldsymbol{\mu}_i + \boldsymbol{\sigma}_i \cdot \epsilon_i$, where $\epsilon_i \sim \mathcal{N}(0, I)$, under a prior $p(Z) = \prod_i \mathcal{N}(0, I)$.

The encoder is a two-layer GCN. A shared first GCN layer produces a hidden representation for all nodes, and two parallel second layers map it to the parameters of a diagonal Gaussian posterior:

$$q_\phi(\mathbf{z}_i \mid X, A) = \mathcal{N}(\boldsymbol{\mu}_i, \mathrm{diag}(\boldsymbol{\sigma}_i^2)), \quad \boldsymbol{\mu} = \mathrm{GCN}_\mu(X, A), \;\; \log \boldsymbol{\sigma} = \mathrm{GCN}_\sigma(X, A).$$

The GCN layers use the symmetrically normalized adjacency $\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ and ReLU nonlinearity; the two heads share the first-layer weights and differ only in the second layer. During training, node-wise latent codes are obtained via the reparameterization trick.

In the original VGAE decoder, the adjacency matrix is reconstructed by modeling each potential edge as an independent Bernoulli random variable, whose probability is obtained from an *inner-product* score. Concretely, with node-level latent representations $Z = [\mathbf{z}_1, \ldots, \mathbf{z}_n]$, the likelihood factorizes as

$$p_\theta(A \mid Z) \;=\; \prod_{i=1}^{n} \prod_{j=1}^{n} p_\theta(A_{ij} \mid \mathbf{z}_i, \mathbf{z}_j), \qquad p_\theta(A_{ij} = 1 \mid \mathbf{z}_i, \mathbf{z}_j) = \sigma(\mathbf{z}_i^\top \mathbf{z}_j),$$

so that higher inner products imply higher edge probabilities (standard logistic Bernoulli model).

Training maximizes the variational lower bound (ELBO),

$$\mathcal{L} \;=\; \mathbb{E}_{q_\phi(Z|X,A)}\Big[\log p_\theta(A \mid Z)\Big] - \mathrm{KL}(q_\phi(Z \mid X, A) \,\|\, p(Z))\,,$$

where q_ϕ is the GCN-based encoder and $p(Z) = \prod_i \mathcal{N}(\mathbf{0}, I)$. In very sparse graphs, to handle the strong class imbalance between edges and non-edges, the authors recommend *re-weighting* the contributions of present edges (or, alternatively, subsampling zeros); in their experiments they adopt the former option while keeping the inner-product Bernoulli decoder.

By treating the presence of an edge between two nodes as a probability, the VGAE decoder becomes a flexible tool for a variety of tasks. The most common use case is *link (edge) prediction*: we want to assign high probability to true (but possibly unobserved) edges and low probability to non-edges. This setting underlies many recommender-system scenarios (e.g., user–item links), knowledge-graph completion, and missing-edge recovery in sparse networks.

Conceptually, one holds out a subset of edges as validation/test links, trains the model to reconstruct the remaining adjacency, and then scores candidate pairs with the decoder (higher scores $\Rightarrow$ more likely edge). Because real graphs are sparse, negatives are not enumerated but sampled: we compare the decoder's scores on held-out positives vs. sampled non-edges and report standard ranking metrics (e.g., ROC-AUC, Average Precision). This protocol directly measures the model's ability to generalize beyond the observed graph.

A second use is *graph generation*. Thanks to the generative formulation, one can produce new graphs by: (i) sampling a latent code for each node from the prior (same prior used at training time), and (ii) decoding edge probabilities between all pairs, optionally sampling the binary adjacency from these probabilities. Practically, one may post-process to enforce symmetry and remove self-loops, and control sparsity via a global bias or threshold on probabilities. This yields synthetic graphs whose connectivity patterns reflect the structure the decoder has learned from data.

These applications, however, are outside the scope of our work. Our goal here is to leverage the same objective as an *in-domain, self-supervised pre-training* signal.

Concretely, we train the VGAE to reconstruct the observed adjacency of the target domain (without labels), thereby shaping node-level representations to encode meaningful structural regularities. The resulting embeddings can then be reused for downstream tasks in the same domain (e.g., node classification, edge regression, ranking), either by freezing them and training a lightweight predictor on top, or by initializing the downstream model with them and fine-tuning end-to-end. In short, we use the VGAE loss not to evaluate link prediction or to generate graphs, but to *pre-train* representations that make subsequent supervised or semi-supervised learning more sample-efficient and stable.

To use this framework as a pre-training technique, we train the encoder to capture the (relational) structure and the temporal dynamics of the graph in a self-supervised manner, using only time-aware subgraphs extracted from the training set[18]. We ask the model to assign high probability to pairs of nodes that are truly connected by an edge in the original graph, and lower probability to pairs that are not. Then, we reuse the encoder's learned weights for the downstream task. Our idea is that this could significantly enhance the model's representational power: by learning to reconstruct (and therefore recognize) the correct links (including the correct relation type, since the graph is heterogeneous) the model will already be able to understand and operate on relational information. All this is achieved without requiring a different dataset or any labels.

Our implementation

Building on prior extensions of variational graph autoencoders to heterogeneous graphs [68], we extend VGAE pretraining to temporal heterogeneous graphs in RelBench, using it as an in domain, time aware, self supervised objective. To the best of our knowledge, no prior work has addressed this specific setting.

Encoder To implement an encoder that works properly with heterogeneous and time-aware subgraphs, we design the encoder as a wrapper class around the original model. This class takes the original model as input and implements an `encode` function that applies the full pipeline till the GNN embedding computation (HeteroEncoder, HeteroTemporalEncoder, GNN), so that we get an embedding vector for each node in the batch[19].

The `forward` method then uses this `encode` function to compute the embeddings for each seed node[20] and projects them through two linear layers: one for the mean

[18]This avoids any potential data leakage.

[19]Those returned by the `NeighborLoader`.

[20]I.e., the nodes for which a prediction must be produced.

$\boldsymbol{\mu}$ and the other for the log-variance $\log \boldsymbol{\sigma}^2$ (for numerical stability). We obtain latent samples via the reparameterization trick:

$$\boldsymbol{\sigma} = \exp\Big(\tfrac{1}{2}\log \boldsymbol{\sigma}^2\Big), \qquad \boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad \boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\varepsilon}.$$

A simplified snapshot of the encoder implementation is provided below:

Algorithm 3 Graph–VAE Encoder (pseudocode)

function ENCODE($\mathcal{B},\ \mathcal{T}$)
  $H \leftarrow f_{\text{base}}$.ENCODE_NODE_TYPES($\mathcal{B}, \mathcal{T}$)
  $z_{\text{dict}} \leftarrow \{\}$
  for $t \in \mathcal{T}$ do
    $h \leftarrow H[t]$
    $\mu \leftarrow h\,W_\mu + b_\mu$
    $\log \sigma^2 \leftarrow h\,W + b$
    if training then
      $\varepsilon \sim \mathcal{N}(0, I)$
      $z \leftarrow \mu + \exp\Big(\frac{1}{2}\ \log \sigma^2\Big) \odot \varepsilon$
    else
      $z \leftarrow \mu$
    end if
    $n_{\text{id}} \leftarrow \mathcal{B}[t].n_{\text{id}}$
    $z_{\text{dict}}[t] \leftarrow (z,\ \mu,\ \log \sigma^2,\ n_{\text{id}})$
  end **for**
  return z_{dict}
end **function**

The mini-batch $\mathcal{B}$ is a sampled subgraph that carries per-type node features and the mapping $n_{\texttt{id}}$ from batch indices back to global node ids; only the node types in $\mathcal{T}$ (which contains the list of node types present in $\mathcal{B}$) are processed for efficiency. The base encoder f_{base} is the graph-aware backbone (heterogeneous/temporal/shallow stack in our implementation) that performs message passing over $\mathcal{B}$ and returns hidden node representations $H^{(t)}$ for each $t \in \mathcal{T}$. Two lightweight linear heads map these hidden states to the parameters of a diagonal Gaussian: $\mu^{(t)}$ and $\log \sigma^{2\,(t)}$, which define the encoder's approximate posterior for nodes of type t. In training mode we draw a standard normal noise ε and obtain a latent code by the reparameterization $Z^{(t)} = \mu^{(t)} + \exp(\frac{1}{2}\log \sigma^{2\,(t)}) \odot \varepsilon$; in evaluation we use the mean $Z^{(t)} = \mu^{(t)}$ for deterministic embeddings. The tuple $(Z^{(t)}, \mu^{(t)}, \log \sigma^{2\,(t)}, n_{\texttt{id}}^{(t)})$ is returned per node type so that downstream decoders can align latent vectors with global node indices when forming edge pairs.

Decoder The decoder proposed by Kipf and Welling [66] is very simple and appropriate for the specific setting they considered, but not for ours:

- Implementing the decoder as an inner product enforces a *symmetric* score and thus loses the direction of the relationship. In fact, because the dot product is commutative, the model cannot represent different probabilities for $i \to j$ and $j \to i$[21]. This assumption is appropriate in the original work, which deals with undirected edges, but it is not appropriate for our setting.
- It does not account for relation-specific semantics (edge types): it simply estimates whether a link exists between two nodes, without identifying which specific relation holds between them. This is acceptable in homogeneous, single-relation graphs, but limiting in heterogeneous settings.

To solve the directionality problem, we design a decoder module that for every pair of nodes builds an edge-wise feature vector by concatenating the source and destination embeddings together with two simple vector operations:

$$x_{ij} \;=\; \Big[\, z_i \;;\; z_j \;;\; |z_i - z_j| \;;\; z_i \odot z_j \,\Big] \in \mathbb{R}^{4d},$$

and maps x_{ij} to a *logit* $s_{ij} \in \mathbb{R}$ through a small MLP. The probability of the edge is $p_{ij} = \sigma(s_{ij})$. Because the input is the ordered concatenation $[z_i; z_j; \cdots]$, swapping i and j changes x_{ij}, so the decoder can model directionality when needed; in undirected settings one typically evaluates each unordered pair once. In practice the MLP has input dimension $4d$ and a single output unit, e.g. $\mathrm{MLP}(x) = \mathrm{Linear}_{4d\to h} \circ \mathrm{ReLU} \circ \mathrm{Linear}_{h\to 1}(x)$. During training, the same decoder is applied to positive edges E^+ and to sampled negatives E^-, producing logits that are then used in the binary cross-entropy reconstruction loss.

Algorithm 4 Graph–VAE DECODER (pseudocode)

function DECODEEDGELOGITS(z_{dict}, `edge_index`, `src_type`, `dst_type`)
  $(g_s,\ g_d) \leftarrow$ `edge_index` ▷ global source/destination node ids per edge
  $(Z_s, \_, \_, nid_s) \leftarrow z_{\text{dict}}[\texttt{src\_type}]$
  $(Z_d, \_, \_, nid_d) \leftarrow z_{\text{dict}}[\texttt{dst\_type}]$
  $\text{map}_s \leftarrow \textsc{Dict}(nid_s[i] \mapsto i)$; $\text{map}_d \leftarrow \textsc{Dict}(nid_d[i] \mapsto i)$
  $i \leftarrow [\,\text{map}_s[g_s[e]] \text{ for } e\,]$; $j \leftarrow [\,\text{map}_d[g_d[e]] \text{ for } e\,]$
  $z_i \leftarrow Z_s[i]$; $z_j \leftarrow Z_d[j]$
  $x \leftarrow \textsc{Concat}\big(z_i,\ z_j,\ |z_i - z_j|,\ z_i \odot z_j\big)$
  $\ell \leftarrow \mathrm{MLP}(x)$ ▷ one scalar logit per edge
  return ℓ
end **function**

[21] Equivalently, computing $A \cdot B$ or $B \cdot A$ yields the same result, i.e., $z_i^\top z_j = z_j^\top z_i$.

For a batch of candidate edges, the decoder first aligns each global node identifier in `edge_index` with its row in the latent matrices by building a fast lookup from global ids to batch indices for the source and destination node types. It then gathers the corresponding latent vectors (z_i, z_j) for every edge and forms a feature vector by concatenating four components: the two endpoints, their absolute difference, and their elementwise product. This $4d$-dimensional edge representation is fed to a multilayer perceptron, implemented as a stack Linear($4d \to h$)–ReLU–(Linear($h \to h$)–ReLU)$^{L-2}$–Linear($h \to 1$), which returns a single logit per edge. These logits are used as Bernoulli parameters (after a sigmoid) in the reconstruction loss for link prediction/training.

Edge sampling function We implement a routine that, given a mini-batch and a specific relation type (src $\xrightarrow{r}$ dst), extracts *positive* and *negative* edges[22]. Positives are obtained by reading the batch adjacency for the selected relation and retaining only those edges whose endpoints are included among the nodes sampled in the batch (i.e., seed and neighborhood nodes). This ensures that every positive edge refers to nodes for which the encoder has provided latents in the current forward pass. Negatives are generated by *corrupting* one endpoint of each positive: for a positive (i, j) we sample a replacement node $\tilde{j}$ from the set of observed destination-type nodes in the batch (or symmetrically a replacement $\tilde{i}$ on the source side), yielding a candidate $(i, \tilde{j})$. Candidates that coincide with real positives are discarded and re-sampled until a valid non-edge is obtained; duplicates among negatives are optionally removed. By default we draw $|E^-| = |E^+|$ negatives for class balance, although a configurable ratio $\rho = |E^-|/|E^+|$ could be used. The function returns two tensors $\texttt{pos\_edges}, \texttt{neg\_edges} \in \mathbb{N}^{2\times E}$ (first row: source indices; second row: destination indices) aligned with the batch's local indexing, ready to be scored by the decoder and consumed by the binary cross-entropy reconstruction loss. To avoid temporal leakage, negative endpoints are sampled only among nodes observed up to $T_\star$.

Loss function The encoder defines a diagonal-Gaussian posterior for each node, $q_\phi(z_i \mid \cdot) = \mathcal{N}\big(\boldsymbol{\mu}_i, \mathrm{diag}(\boldsymbol{\sigma}_i^2)\big)$, while the prior is the standard normal $p(z_i) = \mathcal{N}(0, I)$. The KL between them has the closed form

$$\mathrm{KL} = \sum_i \frac{1}{2} \sum_k \Big(\mu_{ik}^2 + \sigma_{ik}^2 - \log \sigma_{ik}^2 - 1 \Big).$$

This term serves three purposes: it *regularizes* the encoder to avoid representations that only fit the training edges, it *structures* the latent space into a well-behaved,

[22]By *positive* we mean existing edges that appear in the batch for the chosen relation; by *negative* we mean non-edges (false edges) that are not present in the original graph.

approximately "normal" manifold (improving controllability and interpolation), and it *enables generation* by aligning the posterior with a simple prior from which we can later sample.

The objective minimized in practice is the negative ELBO with an optional weight β on the KL term,

$$\mathcal{L} \;=\; \mathcal{L}_{\text{recon}} \;+\; \beta\,\text{KL},$$

often using $\beta = 1$ or a short warm-up. Latent codes are sampled with the reparameterization trick, $z_i = \boldsymbol{\mu}_i + \boldsymbol{\sigma}_i \odot \epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0, I)$. In our implementation, $\mathcal{L}_{\text{recon}}$ is computed on the batch's (time-consistent) positives and on negatives drawn only from nodes observed in the same batch, ensuring no leakage and making the model less prone to overfitting the training data.

Pre-training loop The function "train_vgae" performs in-domain, self-supervised pre-training of the encoder by wrapping the original model in a variational head and optimizing a VGAE objective on time-consistent mini-batches. First, we instantiate a `VGAEWrapper`, which reuses the original encoder to produce one embedding per node type and then projects these embeddings into the parameters of a diagonal Gaussian, yielding per-node $\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$; sampling is performed via the reparameterization trick $z = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \epsilon$. We also instantiate an MLP edge decoder that maps the concatenated edge-wise features $[z_i;\, z_j;\, |z_i - z_j|;\, z_i \odot z_j] \in \mathbb{R}^{4d}$ to a *logit* s_{ij}. The optimizer is deliberately restricted to the parameters that matter for pre-training: the encoder pathway of the original model (`model.encoder_parameters()`), the two linear projections of the wrapper ($\boldsymbol{\mu}$ and $\log \boldsymbol{\sigma}^2$), and the decoder; any task-specific heads remain frozen.

At pre-training time, for each mini-batch we select a relation type (src $\xrightarrow{r}$ dst) that is actually present in the batch, then we (i) compute $z, \boldsymbol{\mu}, \log \boldsymbol{\sigma}^2$ for the two node types[23] via the wrapper; (ii) extract *positive* edges E^+ of that relation from the batch and generate an equal number of *negative* E^- by corrupting one endpoint among nodes observed in the same batch; (iii) score positives and negatives with the decoder to obtain $\{s_{ij}\}$; (iv) compute the VGAE loss $\mathcal{L} = \mathcal{L}_{\text{recon}} + \beta\,\text{KL}$, where the reconstruction term is a binary cross-entropy on $E^+ \cup E^-$ and the KL has the closed form for diagonal Gaussians; and (v) backpropagate and update the parameters. Batches that do not contain usable edges for the chosen relation are skipped; a bounded retry loop avoids wasting compute on empty selections.

[23]Note that each edge type in our setting connects two distinct node types.

3.2.3 Data Augmentation via Relational Aware Edge Dropout

In this section we adopt DropEdge[69] for structural regularization and introduce a relation aware variant that independently drops edges per relation type within our temporal pipeline.

The idea is to regularize the model by injecting stochastic noise into the graph structure during training.

In a message passing layer, the hidden state of a node is updated by aggregating messages from its neighbors. With edge dropout, we **randomly remove** each edge with probability p independently and then run the same forward pass on the thinned batch.

This procedure is the **structural analogue of the standard dropout**. Infact, in standard dropout the model zeros internal activations within a layer, which injects noise at the feature level while leaving the neighborhood structure and all paths between nodes unchanged, so the network must rely on distributed representations rather than on a few co adapting units. Edge dropout instead zeros the messages that would travel along selected edges, which injects noise at the topology level and alters the effective receptive field at every forward pass because the set of available neighbors and multi hop paths varies stochastically across iterations. The first major consequence is a reduction of **overfitting** to particular connections and to frequent hub shortcuts. Since certain edges and even entire two hop or three hop routes can be absent in a given iteration, the model is encouraged to discover patterns that are redundantly supported by multiple alternative neighborhoods rather than by a single brittle corridor. The second major consequence is improved robustness to **missing edges** in deployment. Data pipelines often yield incomplete or delayed relations, so a model that has been trained while repeatedly facing randomly thinned connectivity tends to maintain stable predictions when some links are absent at inference time. In short, standard dropout removes activations while preserving the graph, whereas edge dropout removes connections while preserving the task, and this shift from feature noise to structural noise both regularizes dependency on specific links and prepares the model for realistic sparsity in observed graphs.

At validation and test time the intact graph is used and the model tends to perform more robustly since it was trained to cope with partial connectivity. There is no auxiliary objective that asks the model to detect which edges have been removed. The task is unchanged and the absence of dropped messages simply forces the network to rely on whatever information remains.

In practice values of p between 0.1 and 0.3 work well, one can optionally warm up p from zero over early epochs, and on heterogeneous graphs one can use different probabilities per relation so that dense or noisy relations are downweighted while rare or critical ones are preserved. On very sparse graphs p should be small to avoid

excessive disconnections. The net effect, compared to standard dropout, is a shift from feature noise to structural noise that diversifies which neighbors contribute at each iteration, mitigates over smoothing and over squashing in dense regions, and reduces the tendency to memorize idiosyncratic edges or metapaths while keeping the predictive objective exactly the same.

Algorithm 5 Edge Dropout for GNN (pseudocode)

function EDGEDROPOUT(`batch`, p)
  `out` $\leftarrow$ DEEPCOPY(`batch`)
  for all relation r in `out` do
    $E \leftarrow$ NUMEDGES(`out`$[r]$)
    if $E = 0$ then
      continue
    end if
    `keep` $\leftarrow$ BERNOULLI$(1-p)$ mask of length E
    if $\sum$ `keep` $= 0$ then
      set one random entry of `keep` to TRUE
    end if
    `out`$[r]$.`adjacency` $\leftarrow$ `out`$[r]$.`adjacency`$[:,$ `keep`$]$
  end **for**
  return `out`
end **function**

Given a mini batch graph `batch` and a drop probability p, the routine returns a deep copy `out` where, for each relation type r, it samples an independent Bernoulli mask $\mathbf{k} \in \{0,1\}^E$ with success probability $1-p$ over the E edges of r. If $\sum \mathbf{k} = 0$, one random entry of $\mathbf{k}$ is set to 1 to avoid removing the relation entirely. The adjacency of r is then restricted to the kept indices, and any edge features or timestamps must be restricted in the same way. This procedure is applied to all relation types, affects only the current batch, and serves as structural data augmentation and regularization, since each forward pass observes a randomly thinned subgraph. In practice, we use $p > 0$ during training and $p = 0$ during validation and test.

Algorithm 6 Training with Edge Dropout (pseudocode)

```
procedure TRAIN(loader, model, p)
    for all batch in loader do
        if mode = "train" then
            batch ← EDGEDROPOUT(batch, p)
        end if
        ŷ ← model.forward(batch)
        ℒ ← Objective(ŷ, labels in batch)
        UPDATEPARAMS(model, ℒ)
    end for
end procedure
```

The procedure iterates over mini batches provided by the data loader. During training only, it applies `EdgeDropout` with drop probability p to the current batch, yielding a thinned subgraph. The modified batch is then passed through the model to obtain predictions $\hat{\mathbf{y}}$. A task specific objective $\mathcal{L}$ is computed against the labels contained in the batch, and the model parameters are updated by backpropagation and an optimizer step. At validation or test time the dropout branch is skipped (equivalently, $p = 0$), so predictions are computed on the unmodified batch. Applying dropout only during training provides structural regularization and data augmentation, since each epoch observes different random edge subsets.

To explain the rationale behind this strategy, it is important to understand what happens to the network at the level of receptive fields. A node's receptive field is the set of nodes that can influence a target node through paths available within the current number of layers; with edge dropout, this set changes randomly at each iteration. This explains why edge dropout reduces co-adaptation on specific edges and combats over-smoothing: intuitively, with fewer incoming messages per layer in each realization, deeper stacks are less prone to homogenize representations across large connected regions. Moreover, performing the thinning at every iteration yields an implicit ensemble over exponentially many subgraphs; the shared parameters must therefore perform well across *families* of neighborhood realizations, which improves robustness to spurious or missing links and to sampling variance in neighbor loaders.

For very sparse relations or those known to be semantically crucial, setting p_r close to zero avoids disconnecting seeds.

Edge dropout is a structural analogue of standard dropout that injects noise at the topology level, making the receptive field stochastic across iterations and encouraging distributed representations. This method is simple, architecture agnostic, and effective at reducing reliance on specific neighbors or paths, which

can improve generalization. Its main limitations arise in sparse regions or for low degree nodes where excessive dropping may isolate seeds.

3.3 XMetaPath: A Self-Explainable Meta-Path Graph Neural Network

Modern relational learning systems must not only achieve strong predictive performance, but also provide faithful explanations that are native to the computation they perform. To date, our focus has been on building a **robust**, **automated**, and **flexible** predictive baseline. We now move beyond this to develop an **explainable** model that not only matches black-box performance but also makes its reasoning explicit.

As already mentioned, there are multiple types of explainable models for GNNs. Broadly, approaches fall into two wide families: *post* hoc explainers, which extract explanations after training (e.g., subgraph or feature attributions), and *self-explainable* GNNs, whose architectures are designed to expose an explicit rationale (e.g., prototype- or concept-based modules).

Although post hoc methods are widely applicable and often model-agnostic, they can suffer from limited faithfulness, instability under small input perturbations, and explanations that are plausible but not truly causally relevant to the model's prediction.

For these reasons, the model we design is completely **self-explainable** able to perform **local**, as well as **global**, interpretations. In other words, the same architectural components that drive prediction, expose also human-interpretable signals at inference time.

Building a self-explaining pipeline entails a full architectural rethink. Guided by Ferrini et al.[50, 48], we adopt a meta-path–based design as a principled route to self-explainability. With this model we set an ambitious yet concrete goal:

- **Parsimony through curated meta-paths.** In real relational databases, dozens of linked tables induce a heterogeneous graph with a combinatorial number of potential paths for aggregation and message passing. Meta-paths let us focus on the few relations that actually drive the target, pruning irrelevant nodes and edges; by restricting computation to this small, task-driven subset, the model remains substantially lighter *and more sample-efficient*, which is particularly beneficial in low-data/low-label regimes.

- **Explainable model.** Meta-paths make the reasoning of the model explicit: each path is a human-readable sequence of relations (e.g., `Driver` → `Result` → `Race`) along which we aggregate informations. The prediction is built by combining the per–meta-path signals with data-driven weights, so we can state

how much each meta-path contributed (e.g., 'this meta-path mattered most'). Since each meta path carries its own meaning, the model's reasoning comes into view: the prediction rises from a chorus of meta path signals, weighted by the data, so we can tell how much each one spoke. Because the same weights that combine the paths are also reported as the explanation, what we show is faithful to what the model actually used (no post hoc explainable models needed: the model's own computation is enough to motivate the reasoning process). The same internal quantities used at inference time provide faithful explanations at both local, as well as global level. We offer two complementary interpretations. *Local*: for a given instance x, we read the explanation as (i) the relative importance of each meta-path in the final fusion, and (ii) the most influential relation/hop within the top meta-path(s). In practice, we present a short ranked list: which meta-paths supported the prediction for x, and which relation within them mattered most. *Global*: at the dataset level, we summarize the same signals across many instances (e.g., average importance of each meta-path and how often it is selected as top) together with the distribution of influential relations/hops. This yields a concise picture of the relational mechanisms the model relies on overall, without exposing low-level internals.

Using meta-paths, we restrict message passing to task-relevant, human-interpretable relation chains. A compact curated set yields computational efficiency while enabling intrinsic, inspectable explanations.

3.4 Model's details

This section presents a complete, top–down description of the self-explainable model we employ for relational learning on heterogeneous, time-aware graphs.

For now, assume the full set of meaningful meta-paths is given[24]. In our setting, meta-paths follow a constrained schema: they always begin with a relation whose *source* node type coincides with the prediction's target node type; for every subsequent relation, the source type matches the previous relation's destination type; and we exclude self-loops (each relation thus induces a bipartite graph). The core design choice is that **each meta-path is handled by a dedicated GNN**. Each GNN processes the meta-path's hops in **reverse order**, from the last relation back to the first. We also **flip** each relation ($U \xrightarrow{r} V$) to its reverse ($V \xrightarrow{r^{-1}} U$) and always apply a "dst-aggregates-from-src" update on each triple of the meta-path. This ensures a *target-last* schedule: information is first accumulated

[24]We will discuss later how to automatically learn task- and dataset-specific meta-paths.

on distant nodes, and the *final* hop updates the target nodes (e.g., drivers) after all upstream evidence has been incorporated. The rule is uniform across hops (no special cases for forward/backward steps), simplifies degree-based normalization on the updated side, and prevents premature target updates that would miss upstream contributions.

Consider the 2–hop meta-path starting *from* the target node Drivers:

$$(\text{Drivers} \xrightarrow{r_1} \text{Standings}), \quad (\text{Standings} \xrightarrow{r_2} \text{Races}).$$

We process hops in reverse order and flip each relation:

$$(\text{Races} \xrightarrow{r_2^{-1}} \text{Standings}), \quad (\text{Standings} \xrightarrow{r_1^{-1}} \text{Drivers}).$$

Let $H_{\text{Races}}, H_{\text{Standings}}, H_{\text{Drivers}}$ be the initial node embeddings of the entities related to such tables and let $\text{Agg}_r(H)$ denote the (degree–normalized) message aggregation for relation r. The two reverse hops proceed in the following way:

1. **First reverse hop** (Races $\xrightarrow{r_2^{-1}}$ Standings): $H_{\text{Standings}} \leftarrow \text{Agg}_{r_2^{-1}}\Big(H_{\text{Races}}\Big)$. Here, the destination (updated) node type is *Standings*. For each standings node, we aggregate messages from its adjacent *Races* nodes along r_2^{-1}, thereby restricting message passing to the *Races–Standings* neighborhoods defined by this relation.

2. **Second reverse hop** (Standings $\xrightarrow{r_1^{-1}}$ Drivers): $H_{\text{Drivers}} \leftarrow \text{Agg}_{r_1^{-1}}(H_{\text{Standings}})$, using the already–updated H_{Standing} from step 1.

Hence the final driver representation composes the whole chain:

$$H^{\text{out}}_{\text{Drivers}} = \text{Agg}_{r_1^{-1}}\Big(\text{Agg}_{r_2^{-1}}(H_{\text{Races}})\Big) \tag{3.4.1}$$

Eq. 3.4.1 illustrates *why* reverse order + flip is beneficial: (i) it enforces a **target-last** update, so the target (Driver) is updated only after upstream evidence (Race→Standing) has been incorporated; (ii) it preserves a uniform "dst-aggregates-from-src" semantics at every hop, simplifying normalization and implementation; and (iii) it yields faithful meta-path attributions, since the final target embedding truly reflects the *entire* path's contributions. (Processing forward without flipping would update the target too early, missing the influence propagated from distant hops.)

XMetapath class

To ground the discussion, we begin with the top-level class, `XMetaPath`, which orchestrates the pipeline and instantiates all submodules.

`XMetaPath` is initialized with:

- `data` **(HeteroData):** the graph schema and per–node tables (including `tf_dict` with column names and `time_dict` with timestamps); used only to configure encoders and to read edge types.

- `col_stats_dict`**:** the per–column statistics (means, stds, vocab, etc.) indexed by node type and column name.

- `metapaths`**:** a list of X meta-paths following the aforementioned convention (we denote by X the number of meta-paths).

- **Some hyperparameters**.

At inference (and training) time we invoke `forward(batch, entity_table)`: given the heterogeneous mini-batch and the target node type, `XMetaPath` runs a single, coherent pipeline that maps each target *seed* node to (i) a prediction (one logit per seed node) and (ii) its motivation, i.e., a faithful per–meta-path explanation obtained from the very fusion weights used to combine the meta-path embeddings.

In practice, given a `HeteroData` batch and the target entity table, the forward pass first reads the *seed time*[25] from the batch (`batch[entity_table].seed_time`) and produces type-specific node embeddings via the **HeteroEncoder**. Next, the **HeteroTemporalEncoder** refines these embeddings by incorporating the temporal dimension for node types with timestamps (conditioned on the seed time). Finally, **XMetaPath** instantiates a bank of *meta-path-specific propagators* (one **MetaPathGNN** per selected meta-path) to perform relational propagation along each meta-path. Each propagator consumes (x_{dict}, *batch.edge_index_dict*) and returns a meta-path-specific embedding of fixed width D for the target nodes; the result is a tensor of shape $[N, X, D]$ stacked along the meta-path axis, where N is the mini-batch size and X the number of meta-paths.

The per–meta-path embeddings are fed to a **self-attention regressor**. For each target node, the X tokens $T \in \mathbb{R}^{X \times d}$ are first *contextualized* by a Transformer encoder. A final multi-head self-attention then produces data-dependent weights $w(x) \in \Delta^X$ (non-negative and summing to 1). These weights gate the tokens $(T \odot w(x))$, which are sum-pooled; a small MLP maps the pooled vector to the output logit. On request, we return $w(x)$ together with the prediction: since the same $w(x)$ is used to form the prediction, the reported meta-path importances are *faithful by construction.*

[25]The seed time, also called the *prediction time*, specifies when the target node is to be predicted. It is used to filter future data and enforce temporal consistency, preventing information leakage.

MetaPathGNN: a path-specific propagation

Having defined `XMetaPath` and its per–meta-path view, we now zoom into the `MetaPathGNN` module. Intuitively, it turns a single meta-path into a controlled sequence of message-passing steps, aggregating evidence hop by hop and producing one embedding for the target nodes along that path.

For a single meta-path $\mathcal{P} = (s_1 \xrightarrow{r_1} d_1, \ldots, s_L \xrightarrow{r_L} d_L)$ the module maintains two streams for every node type: a frozen copy of the original features x_t (accessible at every hop) and a current hidden state h_t that is updated hop by hop.

Even within this admissible past, timestamps can span very different horizons. It is therefore useful to **weight information by recency**, so that recent interactions contribute more than stale ones, mitigating (i) degree biases driven by old, dense history, (ii) concept drift[26], and (iii) noise from outdated records. Concretely, we apply an exponential decay $e^{-\lambda_\ell \Delta t}$ to edge messages, with a learnable rate λ_ℓ per hop ℓ.

This mechanism is particularly appropriate in domains where relevance naturally fades over time (e.g., user behavior, transactional/market dynamics, fast-changing operational states). By contrast, when the graph is dominated by quasi-static relations (e.g., long-term memberships, physical constraints, canonical metadata) or when timestamps are sparse/low-fidelity, recency weighting may be unnecessary or even distortive; in such cases, uniform weighting is preferable. We therefore implement recency weighting as an *optional* component, enabled only when the dataset exhibits meaningful temporal decay.

If enabled, for each edge $(u \to v)$ used at hop ℓ, the temporal weight is

$$w_{uv} \;=\; \exp\!\Big(-\,\lambda_\ell \,\frac{\Delta t_{uv}}{\text{time\_scale}}\Big), \qquad \Delta t_{uv} \;=\; \max\{\, t(v) - t(u),\, 0 \,\}.$$

where:

- $w_{uv} \in (0,1]$ indicates the **temporal weight**. These are multiplicative coefficient applied to the message sent from u to v. It equals 1 for zero lag and decays monotonically towards 0 as the lag grows.
- $\lambda_\ell > 0$ is the **per-hop decay rate**: a *learned* parameter (one for hop ℓ) controlling how steeply weights decrease with time. Larger $\lambda_\ell \Rightarrow$ stronger preference for recent evidence.
- $t(u)$, $t(v)$ are the **node timestamps**. These are times associated with the source/destination nodes on this edge, drawn from the batch's time dictionary.

[26]Data-generating process changes over time, so the relationship between entities is not stationary. Therefore, when we aggregate along meta-paths, very old interactions can dominate simply by being numerous, even if they reflect an outdated regime.

By construction of the neighbor loader, $t(u) \leq t_{\text{seed}}(v)$, so only causal neighbors are considered.

- Δt_{uv} indicates the **time lag**. It is a nonnegative temporal distance between sender and receiver on that edge. The max(·,0) acts as a safeguard; with causal loading it coincides with $t(v) - t(u) \geq 0$.
- time_scale > 0 is the **unit normalizer**. This can be described as a hyperparameter that rescales time units (e.g., seconds, days) to a convenient range. For example, the half-life (lag at which $w_{uv} = \frac{1}{2}$) is $\Delta t_{1/2} = (\ln 2)\,\text{time\_scale}/\lambda_\ell$.

If temporal weighting is disabled, we set $w_{uv} = 1$ for all edges; in this case, all messages are weighted equally, with no recency preference.

`MetaPathGNN` instantiates one layer, called **MetaPathGNNLayer**, per relation in the meta-path and applies them sequentially under the reverse+flip schedule. We next describe the single-hop layer that performs the destination update.

3.4.1 MetaPathGNNLayer (single-hop update)

Let us now focus on how to manage and apply a *relation-specific* message-passing operation for a single meta-path hop, this is precisely what `MetaPathGNN` implements.

In particular, for each retained hop ℓ with typed relation $(s_\ell \xrightarrow{r_\ell} d_\ell)$, MetaPathGNNLayer operate on the relation-induced (bipartite) subgraph and compute a weighted, degree-normalized aggregation on the *destination* side:

$$\bar{\mathbf{m}}_v = \frac{\sum_{(u \to v) \in E_\ell} w_{uv}\, \mathbf{h}_u}{\sum_{(u \to v) \in E_\ell} w_{uv}}, \qquad v \in d_\ell.$$

Where:

- ℓ: the current hop of the meta-path, with typed relation $r_\ell : \; s_\ell \to d_\ell$.
- s_ℓ, d_ℓ: source and destination node types for relation r_ℓ.
- $E_\ell \subseteq s_\ell \times d_\ell$: the set of directed edges $(u \to v)$ for relation r_ℓ in the (bipartite) subgraph induced by s_ℓ and d_ℓ.
- $v \in d_\ell$: the destination node being updated at hop ℓ.
- $u \in s_\ell$: a source neighbor of v such that $(u \to v) \in E_\ell$.
- $\mathbf{h}_u \in \mathbb{R}^D$: the current hidden embedding of source node u (embedding dimension D).

- $w_{uv} \geq 0$: the edge weight for $(u \to v)$. With optional recency weighting,

$$w_{uv} = \begin{cases} \exp\Big(- \lambda_\ell \, \Delta t_{uv} / \text{time\_scale} \Big), & \text{if temporal weighting is enabled,} \\ 1, & \text{otherwise,} \end{cases}$$

  where λ_ℓ is a learnable decay rate and $\Delta t_{uv} = \max\{\, t(v) - t(u),\, 0 \,\}$.

- $\bar{\mathbf{m}}_v \in \mathbb{R}^D$: the weighted, degree-normalized mean of neighbor embeddings (i.e., the aggregated message arriving at v).

- $\sum_{(u \to v) \in E_\ell} w_{uv}$: the (weighted) in-degree of v, which normalizes the message scale across different degrees.

Followed by a gated residual update:

$$\mathbf{h}'_v = W_\ell \, \bar{\mathbf{m}}_v + (1 - g) \, W_0 \, \mathbf{h}_v + g \, W_1 \, \mathbf{x}_v^{\text{orig}}, \qquad g = \sigma(\gamma).$$

Where:

- $\mathbf{h}'_v \in \mathbb{R}^D$: updated (pre-activation) hidden state of destination node v at hop ℓ.

- $\mathbf{h}_v \in \mathbb{R}^D$: current hidden state of v *before* applying hop ℓ.

- $\mathbf{x}_v^{\text{orig}} \in \mathbb{R}^D$: (time-enhanced) original input features of v used as a frozen prior.

- W_ℓ, W_0, $W_1 \in \mathbb{R}^{D \times D}$: learnable weight matrices mapping, respectively, the message $\bar{\mathbf{m}}_v$, the prior hidden state $\mathbf{h}_v$, and the original inputs $\mathbf{x}_v^{\text{orig}}$ into the hidden space.

- $g = \sigma(\gamma) \in (0,1)$: scalar gate (per hop ℓ) mixing the two residual paths; $\sigma(\cdot)$ is the logistic sigmoid and $\gamma \in \mathbb{R}$ is a learnable parameter. When $g \approx 0$, the update favors $W_0 \mathbf{h}_v$; when $g \approx 1$, it favors $W_1 \mathbf{x}_v^{\text{orig}}$.

After the linear update, we apply nonlinearity and stabilization:

$$\mathbf{h}_v^{\text{out}} \;=\; \text{Dropout}\Big(\text{LayerNorm}\Big(\phi(\mathbf{h}'_v)\Big)\Big),$$

and write back $\mathbf{h}_v^{\text{out}}$ only for the destination nodes v touched at hop ℓ (using their original indices).

The edge set E_ℓ is constructed *after* the reverse-order+*flip* scheduling, so edges always point from sources to destinations ("dst-aggregates-from-src"). We remap global ids to compact local indices per relation to restrict computation to the touched subgraph and avoid leakage. The gate $g = \sigma(\gamma)$ is a single learned scalar per hop.

Fusion and self-explanations (MetaPathSelfAttention)

The model yields X distinct embeddings[27] for each seed node. We then fuse these representations and produce the final prediction from the aggregated result.

After computing the X path-specific embeddings for each target node, the model forms a token sequence $C \in \mathbb{R}^{X \times D}$. A Transformer encoder contextualizes these tokens into $C' \in \mathbb{R}^{X \times D}$, after which a final multi-head self-attention layer computes an attention matrix $A \in \mathbb{R}^{X \times X}$ and produces per–meta-path weights

$$w \;=\; \mathrm{softmax}\Big(\mathrm{mean}_{\mathrm{heads}}\, A\, \mathbf{1}\Big) \;\in\; \Delta^{X},$$

where $\mathbf{1}$ sums over the key dimension. The same w gates the tokens, $C''_i = w_i\, C'_i$, the gated tokens are pooled by summation, and an MLP maps the pooled vector to the final prediction. Because w is returned (on demand) during inference, the per–meta-path importance reported to the user is identical to the weights that causally shaped the prediction.

This module (the MetaPathSelfAttention module) is an original and important addition to our approach with respect to the original method[50], because it not only assess the influence of each meta path on the final decision and reconnects the exact chain of reasoning behind a given prediction, but also decomposes how much each single meta-path contributes to the prediction, so the model does not only decide, it tells its story in a complete way.

A concrete example of explanation. Let us imagine the driver **Mario Bianchi** as our node of interest. The model must estimate the probability that Mario will finish in the top three in the next race. XMetaPath examines the available information channels and decides how much to rely on each of them. In this case, two meta paths are at the forefront:

$$P_1 :\ (\text{Drivers} \to \text{Standings}), \qquad P_2 :\ (\text{Drivers} \to \text{Qualifying} \to \text{Constructors}).$$

The fusion weights show that the model relies mostly on P_1 (for example, $\alpha_1 = 0.70$ and $\alpha_2 = 0.30$). In simple terms, this means that when predicting Mario's outcome, the model focuses primarily on his most recent standings, while information that passes from Mario through Qualifying into Constructors still matters but plays a smaller role.

At this point, the natural question arises: *which standings are actually influencing the prediction?* The model identifies these nodes during the aggregation along P_1. Each *Standings* node connected to Mario contributes with an internal

[27] Where X is the number of meta-paths.

importance that depends mostly on its recency and on how much its features align with the target. This produces a set of normalized internal weights that we can read as an effective influence for each neighbor.

In our example, Mario is connected to three standings nodes s_1, s_2, and s_3. Without diving into formulas, think of them as the standings the model found most indicative of Mario's current trend. Inside P_1, each of them gets a share of voice in the reasoning process. Suppose their approximate shares are

$$s_1 : \ 50\%, \qquad s_2 : \ 30\%, \qquad s_3 : \ 20\%.$$

This tells us that the prediction for Mario is mainly pushed by s_1 and s_2, while s_3 adds a smaller supporting signal. Internally, the model also applies an adaptive gate that regulates how much new information is admitted. In this case, the gate is open enough, so most of the evidence from s_1 and s_2 is retained.

How does this translate into the final prediction. The signal built along P_1 is fused with the longer path P_2 through the meta path attention weights. Since P_1 dominates ($\alpha_1 = 0.70$), and within it s_1 and s_2 have the highest impact, the overall score for Mario increases, leading the model to predict a high probability of finishing in the top three. In other words, *the reasoning is readable*: Mario's recent standings, especially s_1 and s_2, are the main reasons behind the confident prediction, while the information that flows through Constructors contributes but does not change the final conclusion.

This interpretation is faithful because the weights we report, the attention between meta paths and the internal weights along each path, are exactly the values the model used to compute its prediction. They are not a post hoc explanation, they are the actual trace of the reasoning process.

Although it falls outside the scope of this work, these ingredients can be passed to a large language model to produce natural language explanations. By providing the meta path weights, the selected meta paths, and a few exemplar neighbors with their timestamps and influence scores, the language model can generate a short narrative that is easy to understand for non experts. For instance: *"This prediction draws 60% of its support from the Imola Grand Prix standings on June 15, 2025, where driver Mario Bianchi finished first; it then considers the Monza race from May 18, 2025, where he finished second. Finally, it also takes into account the qualifying results of those same races and the relevant constructors' statistics."* This step does not alter the prediction, it simply makes the reasoning more accessible.

3.5 Meta-path selection

To date, we have focused on a self-explainable GNN that operates on a fixed set of predefined meta-paths, but we have not addressed how to learn meaningful

meta-paths yet.

What we are looking for is a robust, flexible, and fully automated solution that performs reliably across scenarios[28] without any human-in-the-loop supervision. In practice, this means a method that is schema-agnostic (able to accommodate different node/edge types and temporal attributes), task-agnostic (classification or regression), and resilient to data idiosyncrasies such as sparsity, noise, and distribution shift. It should require minimal manual tuning, scale to large graphs, and adapt automatically. Therefore, we explicitly exclude domain-expert designed meta-paths: relying on handcrafted priors would undermine our claim of generality and fully automated operation. The proposed solution must address node-level classification and regression across arbitrary relational databases without manual, domain-specific engineering.

To this end, we implemented three complementary, fully automatic approaches to discover task-relevant meta-paths. Although they share the same objective, each approach targets a specific operating regime. It is therefore meaningful to present all three side by side and compare them empirically, in order to clarify when each method is the most appropriate choice.

For all three approaches, we take inspiration from Ferrini et al.[50]. Specifically, our methods use their greedy meta-path construction skeleton as the backbone, while differing in the criterion used to select the next relation to extend the current meta-path.

As discussed in related work, the principal limitation of Ferrini et al.[50] technique for meta-path selection is that it is tailored for binary classification. Meta-path selection is formulated as a weighted multi-instance problem over positive and negative bags and optimized with a logistic loss, which does not directly extend to multi-class or regression settings without ad-hoc reductions (e.g., one-vs-rest) and their attendant drawbacks. While the MPS framework can be generalized beyond binary classification (e.g., by replacing the logistic MIL surrogate with regression or multi-class variants) such a generalization opens a non-trivial design space. One must decide how to form bags and weights, which surrogate to optimize (Huber/MSE vs. pairwise ranking vs. cross-entropy), how to align the search objective with the downstream metric (R^2/MAE/AUROC), and how to control computational cost and overfitting during iterative expansion. Rather than committing to a single instantiation, we evaluate three fully automatic selectors that trade off task-alignment, accuracy, and execution time: (i) a greedy, GNN-in-the-loop scorer (task-aligned and faithful), (ii) an LLM-based greedy variant (simple and data-efficient), and (iii) a reinforcement-learning policy with a task-aligned reward (escaping greedy myopia and encouraging diversity). Presenting all three clarifies

[28] Across databases and tasks.

when each is preferable (data-scarce vs. data-rich, strict compute budgets vs. higher accuracy, stability vs. adaptivity) and avoids over-claiming that a single "natural extension" suffices across databases and tasks.

This selection algorithm works very well and allow to take into consideration not only an optimization algorithm[29] for the call $r^* \leftarrow \arg\min_r L(r)$, but also the actual GNN model predictions, in this way we are able to give a much more robust evaluation of the utility of each single relation in the metapath, and of each metapath in the pool of the selected ones.

We retain the original hop-by-hop meta-path selection skeleton: at each step we score all admissible relations and extend the current meta-path with the best one.

Algorithm 7 MPS-GNN metapath search learning

```
procedure LearnMPS-GNN(G, R, Y, L_max, η)
    mp* ← [ ]
    F1* ← 0
    S ← Y
    A ← 1
    while |mp| < L_max do
        r* ← arg min_{r∈R} L(r)
        if min_{r∈R} L(r) ≥ η_init(r) then
            return mp*
        end if
        mp ← mp + {r*}                                  ▷ append r*
        gnn ← Train(MPS-GNN(mp), G, Y)
        F1 ← Test(gnn)
        if F1 > F1* then
            mp* ← mp
            F1* ← F1
        end if
        A, S ← NewTargets(S, r*)
    end while
    return mp*
end procedure
```

This pseudocode summarizes the original metapath selection introduced by Ferrini et al.[50]. Unlike their approach, we adopt an alternative criterion to choose the current best relation r^*: we set $r^* \leftarrow \arg\min_r L(r)$, where $L(r)$ is defined without resorting to a weighted multi instance classification surrogate and

[29]In the case of the original paper a weighter multi instance classification task.

is therefore not restricted to binary classification. We then specify an extension specific policy to instantiate $L(r)$ and select the next relation:

(1) Greedy selection. For each candidate relation, we train the GNN model for few epochs on the extended path and pick the relation that yields the largest validation improvement on the downstream metric.

(2) LLM based selection. For each candidate, we build a temporally valid textualization of the local subgraph and query an LLM for a task-oriented score; we aggregate node/bag scores and select the highest one.

(3) Reinforcement learning selection. We cast relation selection as a sequential decision making: a policy proposes the next relation, we observe the validated gain with the extended path, and update the policy accordingly (with controlled exploration and an early-stop when the marginal gain is negligible). Diversity and compute constraints are explicitly encouraged.

This preserves the systematicity and universality of the metapath selection process, while aligning the selection rule with the downstream objective and practical constraints.

3.5.1 Extension 1: Greedy Meta-Path Selection by Direct Validation

The original meta-path selection introduced by Ferrini et al.[50] selects the next relation in a meta-path via a *binary* surrogate objective $L(r)$ (weighted multi-instance logistic loss over positive/negative bags). While fast, that surrogate ties selection to binary classification and may introduce a mismatch between the proxy and the task metric used at evaluation time. **Extension 1** preserves the robustness of greedy search while making it *task-agnostic*: at each step, this is done by replaicing the surrogate with the *actual* model trained on each candidate extension for few epochs and select the relation that optimizes the original *validation* metric. This trades additional time and space for a faithful estimate of downstream performance (classification or regression).

We keep the end-to-end pipeline intact (heterogeneous tabular encoding, temporal encoding, meta-path GNN, task head). The only change is the selection rule: instead of

$$r^\star \leftarrow \arg\min_{r \in \mathcal{R}} L(r),$$

where $L(r)$ denotes the *weighted multi-instance* (MIL) *binary* objective used in MPS-GNN, we *instantiate and train* XMetapath on the tentative metapath $\widetilde{mp} = mp \,\|\, r$, score it on the validation split with a user-specified metric $\mathcal{M}$, and set

$$\text{score}(r) = \begin{cases} \max_e \mathcal{M}(\hat{y}^{(e)}_{\text{val}}, y_{\text{val}}), & \text{(maximize)} \\ \min_e \mathcal{M}(\hat{y}^{(e)}_{\text{val}}, y_{\text{val}}), & \text{(minimize)} \end{cases} \qquad r^\star \leftarrow \arg\max_r \text{score}(r).$$

where e indexes the (few) training epochs used for candidate evaluation.

Basically, starting from the current meta-path, we consider all admissible relations that can further extend the current metapath and select the one that maximizes the GNN's validation performance after training for a few epochs.

In practice, we keep the MPS-style, hop-by-hop selection skeleton while aligning evaluation to the downstream task.

Sequentially, we initialize singleton bags[30] and carry them forward as the meta-path grows, ensuring that instances and labels remain aligned. At each hop we enumerate all admissible relations from the current node type and prune trivial loops, immediate backtracking, domain-banned relations. For each surviving relation we perform a viability check by tentatively expanding the bags[31] along that relation and discarding candidates that do not meet a minimum-coverage criterion (`min_bags`); this avoids scoring relations that would provide too little signal. The remaining candidates are then scored by direct validation: for each relation we instantiate `XMetaPath` on the current meta-path extended with that relation, train the model for a fixed budget of epochs (with early stopping), and record the best validation score according to the chosen metric and its polarity (higher- or lower-is-better). We commit the relation that achieves the best validated improvement, update the meta-path and the propagated bags, keep track of the best-so-far pair, and log every tested (path, score) into an accumulator for later ranking. The procedure stops when no viable candidate remains or the maximum length (L_{max}) is reached; finally, we deduplicate the accumulator by path, keep the best score per unique path, and return both the greedy winner and the Top-K paths ranked by the selection metric.

Extension 1 preserves the MPS hop–by–hop backbone but drives each selection with a validated improvement on the downstream metric after briefly training the model on the extended meta–path. This makes the procedure task–aligned: the same objective used at evaluation time is used to decide which relation to append. The approach is simple to implement, reproducible, and naturally accommodates binary, multi–class, and regression tasks by swapping the metric and loss while keeping the selection logic unchanged.

At the same time, the method incurs non-trivial computational cost because each candidate requires a short training run; if reused repeatedly, the validation split can also become a source of overfitting, and the greedy nature of the search may overlook combinations that pay off only at later hops. If each level evaluates

[30] At the beginning each bag contains the node-id of the seed nodes.

[31] By "expanding the bags" we mean taking one hop along the candidate relation r: each current bag B_i (initially a singleton seed) is replaced by $\widehat{B}_i = \{\, u \mid \exists v \in B_i,\ (v \xrightarrow{r} u) \in E \,\}$. The label y_i is kept, instance weights/attentions (if used) are recomputed/normalized on $\widehat{B}_i$, and empty or undersized bags are discarded (respecting direction and any temporal constraints).

at most R candidates for E epochs, with a per-epoch training+validation cost C_{train}, the total cost is $\mathcal{O}(L_{\max} REC_{\text{train}})$. This is higher than the surrogate-based alternative, but yields selection signals that are faithful to the final objective and support both **classification** and **regression**.

Note that this extension procedure does not scale to large datasets with many tables, since the search space of one hop extensions explodes and the computational cost becomes prohibitive.

3.5.2 Extension 2: LLM based selection

Large Language Models (LLMs) have achieved state-of-the-art results for many tasks, such as reasoning[70], answering questions[71], summarizing texts[72], and creating content[73]. These models have made significant strides in many different fields due to their ability to process and generate human-like text[74].

That said, their application to predictive tasks in relational databases remains largely unexplored. As noted by Solanki et al.[51], the primary challenge lies in aligning fundamentally different data structures: textual vs. graph-based.

We believe that, once this representation problem is solved, LLMs could enable accurate and explainable prediction over relational databases as well.

Previous works have tried to use LLMs to make predictions over relational databases[75, 51]. Despite reaching state-of-the-art results, these approaches come with some important drawbacks. For example, Wydmuch et al.[75] developed a fascinating data structure that compresses all the relevant information from the heterogeneous graph induced by the relational database, and this document, together with the task description, is then passed to an LLM to make a prediction. This approach demonstrates that LLMs can be effectively leveraged on RelBench-style tasks, achieving competitive, often state-of-the-art, performance. However, this approach shows two major drawbacks:

- It is not inherently transparent: the prediction is made by a pre-trained model, and it is not easy to assess why a certain prediction has been made.
- The input context window passed to the LLMs could easily exceed the models' context length, as relational databases usually include large amounts of information and many records.

The solution we propose tries to solve both problems. In particular, we try to keep the best of the two architectures (GNNs and LLMs) and allow for a self-explainable model, while maintaining good predictive results and limiting the context data passed to the LLM.

In particular, we decided to adopt LLMs to **guide the meta-path selection**, but, at the same time, use only the information of the meta-path as context data for the LLM.

More specifically, we adopt the document instantiation of Wydmuch et al.[75], but at selection time we *restrict the prompt to the neighborhood reachable along the meta-path under evaluation*. This path-scoped context serves two purposes: it keeps the prompt within the LLM's context window and makes the score attributable to the candidate relation r, because the model is asked to predict using only evidence accessible through $mp \circ r$. We do so for each admissible r. We then choose the relation that maximizes this score.

We still use the XMetapath model for the actual learning and therefore maintain good predictive capability and explainability, while using the LLM only to guide the meta-path selection.

In our LLM–guided variant, we replace the selection of the next best relation r^* with a single API call that evaluates the same candidate extension using a textual view of the local subgraph. Concretely, for each admissible relation we serialize the neighborhood induced by the current meta–path (respecting direction and time constraints) into a compact prompt and ask the LLM for a score indicating the usefulness of that relation. This eliminates the need to train a model for every candidate, reduces GPU cycles to near zero within the selection loop, and, because calls are independent, admits straightforward parallelization across all one-hop extensions. Beyond speed, LLMs can exploit schema semantics and human–readable attributes (names, descriptions, roles) that are otherwise ignored by purely structural surrogates, which is particularly advantageous in data–scarce regimes.

This substitution does not alter the end–to–end pipeline nor the explainable nature of the predictor: the LLM is used only as a scorer to prioritize which relations to explore next. In short, the LLM serves as a lightweight, parallel, and schema–aware selector that accelerates meta-path discovery while remaining compatible with our task–aligned evaluation and final, transparent GNN training.

Document Construction

Following Wydmuch et al.[75], we define a document in JSON format that includes all the relevant information the LLM can use, considering a specific meta-path. This document is then passed as context to the LLM to make a prediction for several validation samples, and by comparing the prediction accuracy we choose the relation that yields the best performance.

We use JSON because prior work shows that JSON performs well as a text representation of tabular data for LLMs[75].

Following Wydmuch et al.[75], for any sampled seed node from the validation set[32] we build a different document. Each document consists of four components.

[32]Note that, for candidate scoring, we deliberately use *only* the validation split. The LLM is

First, we include the task description. This description explains to the LLM the kind of prediction we are trying to compute. For this reason, we simply use a pre-stored task description provided by Robinson et al.[4]. These descriptions are already clear about the specific task (e.g., "For each driver, predict if they will DNF (did not finish) a race in the next 1 month."). The second component in the context document consists of some in-context samples, n_{met} **training samples** that give a conceptual example of the task and of the relational database structure in terms of primary–foreign key relations. For these n_{met} samples the **label** is also provided. The third part of the document is the entity for which the prediction needs to be made. Finally, we append a concise instruction that makes the expected output format explicit. Unlike the original setup, where formatting was left implicit, this yields consistent and well structured answers across calls and simplifies downstream parsing and evaluation.

It's important to notice that the n_{met} training samples are chosen to be balanced across labels.

This process is executed for a specific path to be evaluated, and it is computed for **all the seed nodes of the validation set**; then the predictive capabilities of the LLM are used to assess the informativeness of the path.

In the second and third parts of the document, a **denormalization** process is needed because much of the important information for the task is usually stored along the links between rows. We should therefore follow links from foreign keys to primary keys (as done by Wydmuch et al.[75]); instead, we apply **meta-path–guided denormalization**, meaning that we stop the denormalization process as soon as the paths in the evaluated meta-path end. Basically, we apply denormalization only for the entities that the meta-path traverses through a relation. This yields two important advantages:

- It lets us assess the informativeness of a meta-path: if applying meta-path denormalization for one meta-path yields better predictive results than another, the former is likely more informative for the task and should be selected.

- It limits the context window substantially, since we do not recursively denormalize along all foreign keys present in the schema,[33] without exceeding

employed solely to rank meta–path extensions and never sees test instances or labels, nor does it alter the training process. This keeps metapath selection on the same footing as hyperparameter tuning and prevents data leakage: consulting the test set, or any future information, during selection would amount to cheating and would invalidate the evaluation.

[33]Denormalization is inherently recursive: when traversing foreign keys for an entity A, we often reach a node B of a different type that exposes further foreign keys to consider. This quickly becomes memory–expensive as the number of primary–foreign key relations grows. In our setting, we mitigate this by performing meta-path–guided denormalization: we restrict traversal

LLM's context limit.

It is important to ensure that the in-context samples are time-consistent to avoid data leakage. Therefore, for every in-context example v, we require $t_v < t_p$, where p is the entity for which we want to make the actual prediction.

Overview of the selection process

Algorithm 8 MPS-GNN metapath search learning

```
procedure LearnMPS-GNN(G, R, Y, L_max, η, n_met, higher, task, metric, api_key)
    mp* ← [ ]
    F1* ← 0
    S ← Y
    A ← 1
    while |mp| < L_max do
        r* ← LLM_best_relation(mp, C, n_met, higher, task, metric, api_key)
        if min_{r∈R} L(r) ≥ η_init(r) then
            return mp*
        end if
        mp ← mp + {r*}                                   ▷ append r*
        gnn ← Train(MPS-GNN(mp), G, Y)
        F1 ← Test(gnn)
        if F1 > F1* then
            mp* ← mp
            F1* ← F1
        end if
        A, S ← NewTargets(S, r*)
    end while
    return mp*
end procedure
```

Algorithm 8 outlines our LLM-guided procedure for meta-path discovery. The routine LLM_find_best_relation (defined below) constructs the prompt, invokes the LLM, and returns the next relation to append to the current meta-path. For clarity, the pseudocode is presented in the single-meta-path setting; in our

to the relations specified by the meta-path under evaluation (mp or $mp \circ r$) and cap the number of neighbors per hop. This avoids the combinatorial blow-up of full denormalization, yielding a lighter, size–predictable view that fits within the LLM's context window while remaining aligned with the task.

implementation we use a beam search that keeps the top candidates at each hop and returns multiple meta-paths rather than a single one.

Algorithm 9 LLM_best_relation

```
procedure LLM_FIND_BEST_RELATION(mp, C, n_met, higher, task, metric, api_key)
    bestRel ← None
    if higher then
        best ← −∞
    else
        best ← +∞
    end if
    for all r ∈ C do
        m̃p ← mp ∘ r                                          ▷ try the extension
        preds ← [ ] ;   gts ← [ ]
        for all (target ∈ val_set of node_type) do
            shots ← TakeSamples(G, node_type, n_met)          ▷ balanced
            json ← BuildJSON(G, m̃p, target, n_met, task, shots)
            yhat ← LLMCall(json, api_key)                     ▷ LLM call
            if yhat ≠ None then
                append yhat → preds ;   append Label(G, target) → gts
            end if
        end for
        s ← Metric(preds, gts, metric)
        if Better(s, best, higher) then
            best ← s ;   bestRel ← r
        end if
    end for
    return (bestRel, best)
end procedure
```

We denote by mp the current meta path and by C the set of admissible one hop relation types. The integer n_{met} is the number of training examples used as in context shots for each validation node. The flag $higher \in \{\text{true}, \text{false}\}$ specifies whether larger values of the validation metric are better. The variable *task* indicates the prediction task (classification or regression) and is used by BuildJSON to format the label space and the prompt. The symbol *metric* names the scalar validation metric employed by Metric (for example AUROC for classification or negative MAE for regression). The string *api_key* provides credentials for the LLM call.

The algorithm also uses a handle to the heterogeneous graph G, the target node

type *node_type*, and a validation set $val\_set \subseteq V_{node\_type}$. For each candidate $r \in C$, the tentative extension is $\tilde{mp} = mp \circ r$, where $\circ$ appends r to the tail of mp. The routine TakeSamples(G, *node_type*, n_{met}) returns n_{met} shots selected via stratified sampling to preserve label balance. BuildJSON($G, \tilde{mp}$, *target*, n_{met}, *task*, *shots*) serializes the subgraph around *target* together with the shots into a compact JSON prompt. LLMCall(*json*, *api_key*) queries the LLM with deterministic decoding and returns a prediction *yhat* or `None`. Metric(*preds*, *gts*, *metric*) computes a scalar score on the accumulated predictions and ground truths, and Better(s, *best*, *higher*) compares scores according to the *higher* flag. The vectors `preds` and `gts` store, respectively, the LLM predictions for each validation node and the corresponding ground truth labels. The variables *bestRel* and *best* keep the current argmax or argmin over candidates, initialized to $-\infty$ if *higher* = true and $+\infty$ otherwise.

Algorithm 10 BuildJSON

procedure BuildJSON(G, $\tilde{mp}$, *target*, n_{met}, *shots*)
  shots ← DEORMALIZE(G, *shots*, $\tilde{mp}$)
  ev_t ← CollectNodeNeigh(G, $\tilde{mp}$, *target*)
  ev_S ← Map$\big(x \mapsto$ CollectEvidence(G, $\tilde{mp}$, x), *shots*$\big)$
  doc ← AssembleDocument(*task*, $\tilde{mp}$, ev_t, *shots*, ev_S)
  return Serialize(*doc*)
end procedure

DEORMALIZE applies a metapath-based denormalization to the shotted samples; CollectNodeNeigh extracts the neighborhood along $\tilde{mp}$ for the node for which we want to make the prediction and apply a de-normalization process based on the metapath $\tilde{mp}$. It also ensures that neighborhood's seed time satisfies $t_{\text{ex}} < t_p$, where "p" indicates the node_id for which we want to make the prediction. AssembleDocument packs task info, meta-path string, target block, evidence, and examples into a single JSON schema; Serialize emits the final JSON string for the LLM. This produces a uniform, schema-aware prompt per validation target and enables fair comparison across candidate relations.

Conclusions and limitation for LLM-guided extension The LLM-based selector offers a pragmatic shortcut for meta-path discovery: instead of optimizing a surrogate loss at each hop, we translate the meta-path–induced neighborhood of a validation target into a compact JSON prompt and ask the model for a task-oriented prediction. In practice, BuildJSON performs a meta-path–guided denormalization of the graph, starting from the source entity, it follows the hops in mp, while enforcing temporal consistency by keeping only rows with timestamp

below a cut-off. For each target we include n_{met} training examples that are stratified across labels, and for time-aware data we only sample examples with $t_{\text{ex}} < t_p$.

Because calls are independent across candidates and targets, the procedure parallelizes naturally; it is also model-agnostic: any base LLM with a sufficient context window can be substituted without changing the algorithm. In our experiments we used an open-weights Llama-3–family model (70B, inference-only, 8k context, deterministic decoding), with prompts comprising a compact JSON plus a short instruction.

This extension is particularly attractive in low-data regimes[34] and when schemas carry rich, human-readable attributes: the LLM can exploit names, roles, or textual fields that purely structural surrogates tend to ignore, delivering competitive selection quality without training a scorer for every candidate. It also reduces GPU usage within the selection loop to near zero and yields fast iteration, while preserving the explainable nature of the final predictor, which remains a transparent meta-path GNN trained and validated in the usual way.

At the same time, there are clear boundaries. The approach incurs monetary cost if many large prompts are required, and adherence to a fixed token budget may force aggressive truncation that dilutes signal. Results can be sensitive to prompt design and, unless decoding is fixed and responses are cached, to stochasticity; moreover, the JSON view inevitably abstracts away some structural nuance that a task-aligned GNN might capture better in data-rich settings. In short, the LLM-guided strategy is a strong choice when data are scarce, schema semantics are informative, or rapid exploration is paramount; when abundant labeled data and compute are available, a purely model-based selector (e.g., direct validation with a lightweight GNN or a learned policy) may provide more stable and fine-grained control at scale.

[34]By *low–data regimes* we mean settings in which labeled supervision is scarce or uneven (few labeled nodes per class, pronounced class imbalance, limited validation budget, or cold start entities appearing late in time), often coupled with sparse edges or noisy tabular attributes. In these conditions, training a scorer for every candidate relation is brittle and compute–inefficient: small datasets make inner-loop training unstable, amplify variance across seeds, and encourage overfitting to the validation split.

The LLM–guided extension is preferable precisely because it replaces that inner optimization with few–shot inference. The prompt pairs a compact, meta–path–scoped view of each validation target with a handful of stratified training examples, allowing the model to leverage broad linguistic priors and human–readable fields (names, roles, descriptions) that carry signal even when the labeled graph is tiny. Selection quality remains competitive without fitting a new model per candidate, GPU usage in the selection loop drops to near zero, and iteration is fast; at the same time, the final predictor stays white–box, since we still train and validate a transparent meta–path GNN once the paths have been chosen. In short, when labeled supervision is limited, the LLM acts as an informative prior and a cheap oracle for ranking candidate extensions, making this variant more stable and practical than training-based selectors under the same constraints.

3.5.3 Extension 3: Reinforcement Learning based selection

Reinforcement learning (RL) has achieved great success in a wide range of AI problems[76, 77, 78]. Motivated by these advances, several works have explored using RL for meta-path discovery in heterogeneous graph settings; however, many of them targeted link prediction problems [80, 79, 81]. Our goal, instead, is to employ a reinforcement learning agent to infer the most meaningful relation from a source object, without having to consider[35] all possible extensions of the current meta-paths (as in extension 1).

Using an RL agent, we aim to replace exhaustive train–validate loops with a learned policy that steers evaluation toward promising candidates, reducing the number of costly model training runs.

In this extension, we deliberately avoid relying on external, pre-trained large language models or third-party services. Instead, the policy is learned end-to-end from the task's own validation signal, keeping the pipeline entirely in-house, self-contained, and reproducible, with clear privacy and cost advantages. Crucially, this design is also substantially faster: by steering evaluation toward promising candidates and pruning unproductive branches early, it sidesteps the combinatorial burden of exhaustively training and validating all possible relation combinations, resulting in far fewer model runs, lower compute, and markedly shorter execution time.

Reinforcement Learning architecture

Reinforcement learning models decision making as a **Markov Decision Process** (MDP) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, in which an **agent** interacts with an **environment** to maximize an expected cumulative **reward**. At each timestep t, the agent observes a state $S_t \in \mathcal{S}$, selects an action $A_t \in \mathcal{A}$ according to a policy $\pi(a \mid s)$, the environment transitions to S_{t+1} and emits a reward R_{t+1}.

Formally, the components are:

- $\mathcal{S}$, the (measurable) state space, which denotes the set of all possible states the agent can be in;
- $\mathcal{A}$, the action space, which is the set of all the possible actions an agent can perform;
- $P(\cdot \mid s, a)$, a probability distribution on $\mathcal{S}$ specifying the transition dynamics from s under a;

[35] By "consider" we mean training the XMetapath model and obtaining its validation score.

- $R(s, a, s')$, the expected immediate reward when taking a in s and moving to s', i.e., $R(s, a, s') = \mathbb{E}[\, r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s' \,]$;
- $\gamma \in [0,1)$, the discount factor, used for two complementary reasons:
  1. **Mathematical reason.** Without discounting the sum of rewards can diverge.
  2. **Practical/modeling reason.** The future is uncertain: *in practice we assign less weight to rewards predicted to arrive in the far future than to those in the present or near future, which are more controllable.*

The **Markov property** states that, conditional on the current state and action, the distribution of the next state and reward does not depend on past history: for any $h_t = (s_0, a_0, \ldots, s_t)$ and action a_t,

$$\mathbb{P}(s_{t+1}, r_{t+1} \mid h_t, a_t) = \mathbb{P}(s_{t+1}, r_{t+1} \mid s_t, a_t).$$

We can summarize this formalism by saying that in a MDP *the future is independent of the past, given the present.*

A (stochastic) **policy** maps states to probability. A policy *fully defines the behaviour of an agent* in the environment. In this work we consider *only the discrete case*: states are discrete metapath prefixes and, at each state, the set of admissible actions is finite. Letting $\mathcal{A}(s) \subseteq \mathcal{A}$ denote the legal actions in state s, we write

$$\pi : \mathcal{S} \to \Delta(\mathcal{A}(s)), \qquad \pi(a \mid s) = \mathbb{P}(A_t = a \mid S_t = s),$$

with, for all $s \in \mathcal{S}$,

$$\sum_{a \in \mathcal{A}(s)} \pi(a \mid s) = 1, \qquad \pi(a \mid s) \geq 0 \quad \forall a \in \mathcal{A}(s).$$

In our implementation, $\mathcal{A}(s)$ is the finite set of typed relations admissible from the last destination type of the prefix.

The standard control objective is to maximize the expected discounted return

$$J(\pi) \;=\; \mathbb{E}_{s_0 \sim d_0,\, \pi,\, P}\Big[\sum_{t=0}^{\infty} \gamma^t\, r_{t+1}\Big],$$

and, in finite MDPs with discounted infinite horizon ($\gamma < 1$) or finite horizon, there exists an optimal stationary policy $\pi_\star$ such that $V^{\pi_\star}(s) \geq V^{\pi}(s)$ for all $s \in \mathcal{S}$ and all π.

A key challenge in RL is the reward signal: it provides the agent's primary learning feedback, yet rewards are often **sparse** (many actions yield no immediate signal), **delayed** (benefits materialize only after several steps), and **stochastic** (the

same state–action may produce different outcomes due to noise and uncertainty). Consequently, credit assignment and exploration are central to effective learning, as actions may have long term consequences and it may be better to sacrifice immediate reward to gain more long-term reward (E.g. Refueling a helicopter might prevent a crash several hours later).

Proposed solution

We propose a **model-free** RL approach to learn the most significant meta-paths for a given task and database. In a model-free setting, we do not assume access to the MDP's transition dynamics or reward function. The agent interacts with the environment, taking actions and learning directly from episodes of experience.

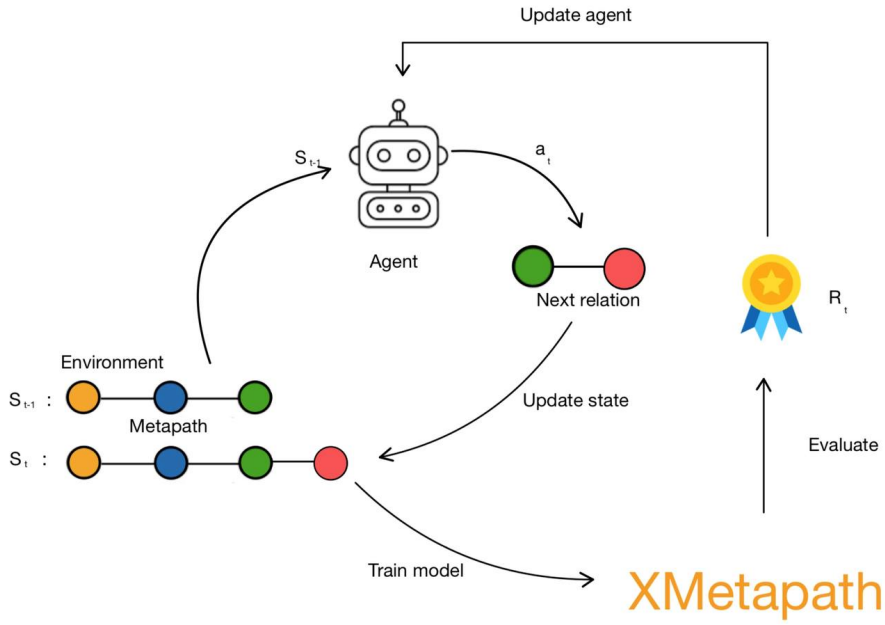


Figure 3.2: An overview of the proposed framework. At each timestep t, the agent receives the current state of the environment (the current metapath) and selects an action (the next relation to add to the metapath). The XMetapath model uses the newly generated metapath as input, and the resulting performance gain is used to update the agent.

More specifically, we use a model-free, tabular, off-policy Q-learning approach with Boltzmann (softmax) exploration.. We select actions according to a Boltzmann/softmax policy over Q-values:

$$\pi_\tau(a \mid s) = \frac{\exp\Big(Q(s,a)/\tau\Big)}{\sum_{b \in \mathcal{A}(s)} \exp\Big(Q(s,b)/\tau\Big)},$$

where the temperature $\tau > 0$ controls the exploration[36]–exploitation[37] trade-off ($\tau\downarrow \Rightarrow$ greedier; $\tau\uparrow \Rightarrow$ more exploratory). We anneal τ down in the final phase.

Annealing τ from a larger to a smaller value smoothly shifts the policy from exploration to exploitation. We cast metapath construction as a finite-horizon decision process in which the agent extends a metapath one relation at a time and learns from actual apisodes.

A **state** $s \in \mathcal{S}$ is the current metapath prefix, i.e., an ordered list $s = [e_1, \ldots, e_\ell]$ with $e_i \in E$; the initial state is $s_0 = \varnothing$. Given s, the set of *legal actions* (admissible relations that can extend the prefix) is

$$\mathcal{A}(s) = \left\{ a = (\mathrm{src}, \mathrm{rel}, \mathrm{dst}) \in E \;\middle|\; \begin{array}{l} \mathrm{src} = \mathrm{last\_dst}(s), \\ \mathrm{dst} \neq \mathrm{start\_type}, \\ \mathrm{dst} \notin \mathrm{visited\_dst}(s) \end{array} \right\}.$$

An **action** is any $a \in \mathcal{A}(s)$. Applying a induces the *transition*

$$s' = \begin{cases} s \oplus a, & \text{if the extension has sufficient support (valid bags)} \\ & \text{and passes early-stop checks,} \\ s, & \text{otherwise (penalized no-op).} \end{cases}$$

We model metapath construction as an **episodic** MDP. An *episode* starts from an initial state s_0 and ends upon reaching a terminal condition.

Episodicity is *not* a requirement of MDPs. In *continuing* tasks there are no terminal states and the process runs indefinitely.

Our problem, instead, is naturally episodic because each episode constructs a single metapath and then stops according to a termination rule.

Each episode starts at the empty prefix $s_0 = \varnothing$ and terminates if one of the following holds:

1. **Length limit:** $|s| = L_{\max}$. We provide the metapath discovery routine with a hyperparameter $L_{\max}$ that limits the metapath length, i.e., the maximum number of relations allowed in a metapath. This limit is particularly useful when overly long metapaths are known to degrade model performance (or inflate computation), as it constrains the search to plausible candidates and improves stability.

[36] *Exploration* is the purposeful selection of uncertain or seemingly suboptimal actions in order to broaden the search space and discover potentially better solutions.

[37] *Exploitation* is the choice of the action that currently appears most valuable, aiming to maximize the expected return based on existing knowledge.

2. **No legal actions:** $\mathcal{A}(s) = \varnothing$. This can occur because, at each iteration, we exclude from the set of possible extensions any relation $a = (\text{src}, \text{rel}, \text{dst})$ for which (i) src $\neq$ last_dst(s) (type mismatch), (ii) dst = start_type (return-to-start forbidden), or (iii) dst $\in$ visited_dst(s) (no revisits). Since candidates are drawn from the actual typed edges exposed by `edge_index_dict`, if the last destination type has no outgoing relations satisfying these constraints, the admissible set becomes empty and the episode terminates. (Practically, candidates that yield insufficient support, as in the case in which there are too few valid bags, are penalized and skipped; repeated rejections can also exhaust the viable options at this step.)

3. **Early stop / patience:** we terminate the episode if the last extension produces a *relative degradation* larger than ε for higher-is-better metrics: $J(s') < (1 - \varepsilon)\, J(s)$; for lower-is-better: $J(s') > (1 + \varepsilon)\, J(s)$. Otherwise, we keep a counter of consecutive *non-improvements*, where a step counts as a non-improvement if the marginal gain does not exceed a small tolerance $\delta > 0$: for higher-is-better, $\Delta J := J(s') - J(s) \leq \delta$; for lower-is-better, $J(s) - J(s') \leq \delta$. The counter is reset whenever an improvement occurs. When the counter reaches the patience budget P, the episode is stopped.[38]

It is important to note that the *patience*-based early stopping decouples the nominal horizon $L_{\max}$ from the *effective* metapath length chosen by the agent. At step t, let $\Delta J_t := J(s_{t+1}) - J(s_t)$ (flip the sign for lower-is-better metrics). We continue extending the prefix while $\Delta J_t > \delta$ and no relative degradation beyond ε occurs; otherwise we increment a counter of consecutive non-improvements. The episode stops when the counter reaches the patience budget P (or when $\mathcal{A}(s_t) = \varnothing$), yielding the stopping time and the *effective* length $L^\star := |s_\tau|$. Consequently, $L_{\max}$ can be set to a very large value (even conceptually unbounded) without over-committing to long paths: the agent will *automatically* stop at the length that is most beneficial given the data and task. This is particularly useful when the optimal metapath length is dataset- and task-dependent; rather than hand-tuning $L_{\max}$ per setting, **we let the procedure select $L^\star$ adaptively via the (ε, δ, P) early-stop mechanism by setting L_{max} to an high (or infinite) value**.

Reward (marginal gain). Each step appends one relation to the current metapath. The environment responds by briefly training XMetapath on the candidate path and measuring the task's validation metric. The per-step reward is the *metric*

[38] In our final configuration we set $P = 3$. That is, if for three consecutive iterations the relation chosen by the agent does not increase the validation score by more than δ (and does not violate the degradation test above) we stop the episode.

improvement induced by the chosen extension (for higher-is-better metrics we use the metric as is; for lower-is-better metrics, e.g., MAE, we negate it) so that larger means better. Intuitively:

> *"If adding this relation improves validation, give a positive reward; if it worsens it, give a negative reward."*

In practice, after tentatively extending s with a and briefly training/evaluating the downstream model, we obtain a validation score $J(s \oplus a)$. We define a step-wise *marginal* reward:

$$r(s,a) \;=\; \begin{cases} J(s \oplus a) - J(s), & \text{if } J \text{ is higher-is-better,} \\ J(s) - J(s \oplus a), & \text{if } J \text{ is lower-is-better.} \end{cases}$$

If the extension yields too few valid bags, we assign a penalty $r = -1$ and stay in the same state $s' = s$.

Two practical adjustments improve robustness and budget control: (i) *invalid or weak expansions* (e.g., too few valid bags or violated constraints) receive a fixed penalty in lieu of training; (ii) a small *per-hop length penalty* encourages compact, informative meta-paths under a fixed compute budget.

Q-learning update (bootstrapped). We maintain a tabular action-value function $Q(s,a)$, where $Q(s,a)$ stores, for each encountered state–action pair (s,a), a **single number**: the **estimate of the action value**, i.e., the expected (discounted) return obtained if one takes that action in that state. We update these value with the standard off-policy Q-learning rule:

$$Q(s,a) \;\leftarrow\; (1-\alpha)\,Q(s,a) \;+\; \alpha\Big[\, r(s,a) \;+\; \gamma \max_{a' \in \mathcal{A}(s')} Q(s',a') \,\Big],$$

where $\alpha \in (0,1]$ is the learning rate and $\gamma \in [0,1]$ the discount factor.

Warm-up (agent pretraining). Before the final discovery of top-X metapaths, we run a *warm-up* phase in which a single RL agent is trained over multiple short episodes to populate its Q-table with informative, data-driven priors. Each warm-up episode starts from the empty prefix and proceeds step-by-step exactly as in the main loop (same legality constraints, same marginal reward, same Q-learning update), but with *cheaper evaluations* and *more exploration*: (i) we train/evaluate the downstream model for only a few epochs (fast, noisy proxy of the validation signal), (ii) we use a higher temperature τ in the Boltzmann policy to diversify action selection, and (iii) we keep the same termination rules (length limit, no legal actions, early-stop/patience). We use a single agent shared across episodes (so its

Q-table accumulates experience), with warm-up settings such as `num_episodes`= 5, `epochs`= 5, and a relatively high temperature ($\tau \approx 1.0$) together with a faster Q-learning rate ($\alpha = 0.5$). After warm-up, we *always* anneal the temperature to promote exploitation and we *reduce* the learning rate for stability, before launching the final RL episodes and the Top-x selection.

The warm-up has three main goals: (1) *cold-start reduction*: initializing $Q(s, a)$ from actual episodes avoids starting from a flat table (all zeros) and gives the policy a sensible shape before the more expensive search; (2) *variance reduction*: multiple (cheap) episodes average out the noise of short trainings, yielding more reliable action preferences per state; (3) *dataset-specific priors*: the agent quickly adapts to the graph's local statistics (e.g., which relations tend to create well-supported bags) without committing to long, costly episodes. Importantly, warm-up does *not* decide the final metapaths: we discard the candidate registry collected during warm-up and only retain the learned Q-values; the final phase then re-runs episodes with lower τ and fuller evaluations to collect *full-length* candidates and perform the Top-X diverse selection.

Top-X diverse metapaths (selection procedure). After the final RL episodes, we obtain a registry of candidate metapaths with their best validation scores. Let $\mathcal{C}$ be the set of *unique* candidates and let $S(p) \in \mathbb{R}$ denote the quality score of p (the best validation metric observed for p across episodes; sign-flipped if the metric is lower-is-better). Our goal is to return a set $\mathcal{X} \subseteq \mathcal{C}$ that is both high-scoring and diverse.

We rule out trivial redundancy by forbidding the selection of a metapath together with any of its *extensions* (or, symmetrically, any *prefix* of it). Let a path be a finite sequence of typed relations:

$$p = (e_1, \ldots, e_L), \qquad q = (f_1, \ldots, f_M), \quad e_i, f_j \in \mathcal{E}.$$

We say that p is a (proper) **prefix** of q, and write $p \prec q$, if

$$L < M \quad \text{and} \quad e_i = f_i \ \text{ for all } i = 1, \ldots, L.$$

The **no-prefix constraint** requires that the returned set $\mathcal{X}$ contains no two paths in a prefix relation:

$$\forall\, p \neq q \in \mathcal{X} : \quad \neg\big(p \prec q \ \vee \ q \prec p\big).$$

During greedy selection, before adding a candidate p we check whether there exists $q \in \mathcal{X}$ such that $p \prec q$ or $q \prec p$; if so, we *skip* p. This filter is applied prior to the MMR scoring step.

With this check we prevent returning both a pattern and its trivial extension (e.g., $A \to B$ together with $A \to B \to C$), which would convey nearly the same semantic signal and reduce diversity.

Among the remaining candidates, we apply a Maximal Marginal Relevance (MMR) criterion that trades off score and dissimilarity. Let $\mathrm{sim}(p, q) \in [0,1]$ be a similarity between paths (we use a *prefix-overlap* similarity):

$$\mathrm{sim}(p, q) \;=\; \frac{\#\{\text{matching positions from the start}\}}{\min(|p|, |q|)} \quad .$$

Given a diversity weight $\lambda \in [0,1]$, at each iteration we add to $\mathcal{X}$ the candidate that maximizes

$$\mathrm{MMR}(p \mid \mathcal{X}) \;=\; S(p) \;-\; \lambda \cdot \max_{q \in \mathcal{X}} \mathrm{sim}(p, q),$$

subject to the no-prefix constraint. λ controls the quality–diversity trade-off ($\lambda = 0$ reduces to top-X by score; larger λ promotes dissimilarity).

Conclusions (RL-based meta-path selection). The reinforcement-learning variant casts meta-path construction as a finite-horizon decision process: the state encodes the current prefix, actions are the admissible relations (plus an optional STOP), and the reward is the validated improvement yielded by the extension. A simple tabular Q-learning agent with Boltzmann exploration learns, across short episodes, to steer evaluation toward promising candidates, while early stopping and a final diversity-aware Top–X selection (e.g., via MMR) prevent redundant paths and keep the outcome interpretable. In practice, this replaces many exhaustive train–validate loops with a policy that focuses compute where it matters, remains fully in-house (no external services), and preserves the white-box nature of the final predictor, which is still a transparent meta-path GNN trained and validated in the usual way.

This design works best when the search space is large or unevenly informative, so that blindly training a candidate per hop is wasteful. By reusing experience across episodes and adaptively balancing exploration and exploitation (via temperature annealing), the agent typically requires fewer evaluations to surface high-quality and diverse meta-paths, and it naturally adapts the effective length through patience-based early stopping rather than a hard, problem-specific choice of $L_{\max}$. It is also a sensible choice when privacy, cost, or platform constraints rule out LLM-based scoring, since the entire loop depends only on the task's own validation signal.

At the same time, RL introduces its own trade-offs. The reward can be noisy (short training budgets), credit assignment is non-trivial, and performance can be sensitive to hyperparameters such as learning rate, discount, temperature, and patience; heavy reuse of a single validation split also calls for care (e.g., nested validation) to avoid selection bias. In very small search spaces, or when evaluation per candidate is cheap and stable, a direct-validation greedy/beam search may be simpler and just as effective. Overall, the RL-based selector is a compelling

middle ground: it avoids the monetary and context-window constraints of LLM scoring, reduces redundant computation compared to exhaustive baselines, and yields diverse, high-quality meta-paths, provided that the reward is stabilized, invalid actions are masked, and the stopping criterion is well tuned to the task and dataset.

Chapter 4

Experiments

In this chapter we present an empirical evaluation of the proposed models on RelBench. We compare backbones under a unified pipeline and study the effect of self-supervised pretraining strategies. All experiments are run with three random seeds, and we report the mean and standard deviation across seeds.

Aims and Hypotheses

Our experiments evaluate three hypotheses. First, relational deep learning models tend to outperform the non relational baseline across tasks and datasets, with the largest gains precisely in tasks where the relational content is central to prediction rather than peripheral. When the target mostly depends on interaction patterns, multi hop dependencies, and cross entity constraints, relational inductive bias and message passing improve learning by integrating complementary signals across neighbors and relations, capturing higher order structure, and filtering noise. By contrast, in regimes dominated by strong per entity attributes, where most predictive information is already local, the advantage narrows and the graph contributes only a small incremental signal. This perspective explains where we expect the largest improvements and aligns with our empirical observations.

Second, pretraining yields a mild, yet consistent, improvement on downstream performance. The gains are not dramatic on any single task, but they persist across tasks and seeds, indicating a stable contribution to generalization.

Third, XMetaPath achieves competitive accuracy while providing explanations that remain faithful to the underlying computations. Because XMetaPath is meta-path based and self-explainable, it produces both local and global explanations: a map of the exact relational routes that carried information to a decision and a summary of which patterns matter across the dataset. Explanations are not an afterthought, they are a first class output, inseparable from the prediction itself.

In settings where interpretability is required, from auditing and accountability to high stakes operations with a human in the loop, such transparency turns the model into a tool that can be relied on in practice. Yet this clarity does not come at the expense of predictive power. By selecting and weighting relational paths with care, the model can rival black box alternatives, and in some cases surpass them, precisely because the inductive bias keeps attention on the signals that truly matters through the graph. What the model says it used is what it actually used, and the path it traces is the path the data followed.

We follow an evaluation protocol that respects temporal causality and uses fixed validation rules. All models share the same data splits, the same tuning budget, and the same early stopping procedure.

We follow the RelBench[4] reference exactly in both evaluation and reporting. For **node regression** tasks we use *Mean Absolute Error* (MAE) and we **always print three decimal digits** (e.g., 3.870), matching the RelBench tables. For **node classification** tasks we use *ROC AUC* and we **always print two decimal digits** (e.g., 75.40), again as in RelBench. This fixed formatting makes our tables directly comparable to the reference[4]. As usual, we annotate with ↑ metrics to be maximized (ROC AUC) and with ↓ metrics to be minimized (MAE).

For node regression, following the RelBench evaluation protocol, we report Mean Absolute Error (MAE), defined for ground truth y_i and predictions $\hat{y}_i$ on n evaluation nodes as:

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

For MAE, lower values indicate better predictions. The score is expressed in the same unit as the target, so it can be read directly: if the target is lap time in seconds, a MAE of 0.250 means the predictions are off by a quarter of a second on average. To keep results strictly comparable, throughout the tables we report MAE with three decimal digits, exactly as in RelBench.
For binary classification we evaluate how well the model ranks positive nodes above negative ones using the area under the receiver operating characteristic curve (AUC-ROC). Each sample receives a real valued score s_i and we conceptually sweep a decision threshold t across the entire range to trace the trade off between true positives and false positives. The rates are:

$$\text{TPR}(t) = \frac{\text{TP}(t)}{\text{P}}, \qquad \text{FPR}(t) = \frac{\text{FP}(t)}{\text{N}}.$$

Here, TP(t) denotes the number of true positives at threshold t, that is the instances correctly classified as positive. FP(t) denotes the number of false positives, that is the negative instances that are incorrectly classified as positive. P and

N represent the total number of positive and negative instances in the dataset, respectively.

Consequently, the True Positive Rate (TPR) corresponds to the proportion of correctly identified positive cases over all actual positives, while the False Positive Rate (FPR) measures the proportion of negatives that are mistakenly classified as positive.

The area under the resulting curve equals the probability that a randomly drawn positive node receives a higher score than a randomly drawn negative node,

$$\mathrm{AUC} = \Pr\left(s^+ > s^-\right) + \tfrac{1}{2}\Pr\left(s^+ = s^-\right).$$

Values range from 0.50 for random ranking to 1.00 for perfect ranking. The measure is 'threshold free' because it aggregates performance over all possible decision thresholds rather than fixing one. Moreover, since it depends only on the ordering[1], not on the absolute values, ROC AUC is typically stable under class imbalance, which is common in our graphs. The metric evaluates ranking quality rather than probability calibration.

4.1 Model selection

Table 4.1: Comparison on RelBench tasks. LightGBM and HGraphSAGE numbers are those reported in the RelBench paper [4]. ↑ means higher is better, ↓ means lower is better. Best results are highlighted in **bold**, as in the subsequent tables.

	LightGBM	**HGraphSAGE**	**HTG**	**HGAT**
REL F1				
Driver Top 3 ↑ (ROC AUC)	73.92 ± 5.75	75.54 ± 0.63	75.49 ± 2.91	**78.33 ± 2.14**
Driver Position ↓ (MAE)	4.170 ± 0.137	4.022 ± 0.119	**3.872 ± 0.151**	3.983 ± 0.148
Driver DNF ↑ (ROC AUC)	68.56 ± 3.89	72.62 ± 0.27	**75.35 ± 1.33**	73.47 ± 1.07
REL TRIAL				
Study Adverse ↓ (MAE)	**44.011 ± 0.998**	44.473 ± 0.209	47.809 ± 0.417	48.846 ± 0.302
Study Outcome ↑ (ROC AUC)	**70.09 ± 1.41**	68.60 ± 1.01	67.31 ± 2.64	67.07 ± 1.72
Site Success ↓ (MAE)	0.425 ± 0.003	0.400 ± 0.020	**0.346 ± 0.091**	0.360 ± 0.052
REL AVITO				
User Clicks ↑ (ROC AUC)	53.60 ± 0.59	65.90 ± 1.95	65.98 ± 2.34	**67.23 ± 2.02**
User Visits ↑ (ROC AUC)	53.05 ± 0.32	66.20 ± 0.10	**69.00 ± 1.35**	66.69 ± 1.61
Ad Ctr ↓ (MAE)	0.041 ± 0.000	0.041 ± 0.001	**0.039 ± 0.003**	**0.039 ± 0.003**

[1]Note infact that this metric is equals to the probability that a randomly chosen positive receives a higher score than a randomly chosen negative, $\mathrm{AUC} = \Pr(s_+ > s_-)$. Any strictly increasing transformation of the scores preserves this ordering and leaves AUC unchanged. The ROC curve aggregates true positive rate and false positive rate across all thresholds; both are rates normalized by the size of their classes, so AUC is typically insensitive to class imbalance.

We compare four models. (1) A non relational baseline based on LightGBM[12], which treats RelBench as a tabular prediction problem and does not perform any explicit message passing[2]. The reported results for this model are directly taken from the RelBench paper [4], where LightGBM serves as a strong non relational reference point. (2) A relational deep learning baseline, *HGraphSAGE*, proposed by Robinson et al.[4]; for this model we report the results exactly as released in their work [4], which we use as our reference. (3) Our HeteroTemporal Graphormer (HTG). (4) Our Heterogeneous Graph Attention Network (HGAT). All models are evaluated under the same conditions, using the identical data processing pipeline.

The only difference in the training of the models is that, due to its memory requirements, HTG was trained with a smaller batch size and neighbor fanout due to attention memory requirements. In our default setup we use batch size 512 and neighbor fanout 256, whereas for HTG we use batch size 32 and neighbor fanout 16; this can cap additional gains.

REL F1. Averaged across the three tasks, our attention–based models (HTG and HGAT) improve over the tabular baseline by $\approx 7.6\%$ and over the RDL reference (HGraphSAGE) by $\approx 3.7\%$.[3] The gains are consistent: HTG alone is slightly below HGraphSAGE on *Driver Top 3*, but HGAT closes the gap and sets the best score, while both models improve *Driver Position* and *Driver DNF*.

REL TRIAL. Performance is clearly task–dependent. The picture is mixed: our models excel on *Site Success* (MAE reduced by 19.3% vs. LightGBM and 14.3% vs. HGraphSAGE), but the tabular baseline remains stronger on *Study Outcome* and *Study Adverse*.

In this database, a large fraction of the predictive signal resides in the textual attributes of individual nodes. When a task depends on these fine grained local details, relational aggregation has limited marginal value. Put differently, when the question is "what kind of trial is this, how is it designed, what criteria and outcomes does it describe," the answer lies in the text fields attached to the specific node rather than in its neighborhood. In such settings, the incremental value of relational modeling in RDL over competitive tabular models is expected to be modest and can even be negative when textual features already capture most of the signal.

This perspective helps explain why, on Study Adverse and Study Outcome,

[2]LightGBM results are taken from RelBench. In this study, LightGBM is an automatic non relational baseline: for entity classification and regression it is trained only on the raw features of the single target entity table. No manual feature engineering is used; training and evaluation are performed using a LightGBM architecture[12].

[3]Percentages are computed per task as relative improvement of the better of HTG/HGAT with respect to the baseline, using higher–is–better for ROC AUC and lower–is–better (relative error reduction) for MAE.

non relational baselines such as LightGBM remain ahead. For these tasks the decisive signals are concentrated in local textual attributes of nodes, and the graph does not add stable or reusable context, which can even make relational models more prone to overfitting. In contrast, on Site Success the situation reverses: here relational information is essential, since site performance reflects patterns shared across studies, sponsors, facilities, and eligibility populations, which can only be captured by graph aggregation.

REL AVITO. Here the advantage is pronounced. Our models attain the best score on all three tasks, with average improvements of $\approx 20.1\%$ over LightGBM and $\approx 3.7\%$ over HGraphSAGE; HGAT leads on *User Clicks*, HTG on *User Visits*, and both reduce MAE on *Ad CTR*. [4]

Overall, the evidence points to a clear dependency on the nature of the task. When the target hinges on relational structure, models that aggregate along the graph unlock substantial gains; when the signal is concentrated in rich local attributes, especially text fields attached to a single node, tabular models remain difficult to surpass. In the few settings where we do not observe improvements, the pattern is consistent with a regime in which the decisive information resides within the node itself rather than in its connections, leaving limited headroom for additional relational computation to add value.

Equally important, the value of relational modeling is not only about surpassing tabular baselines on test performance. It lies, above all, in making relational reasoning **automatic** and **systematic**. Instead of hand crafting features for every new domain, curating joins by intuition, and encoding expert knowledge case by case, a single well designed relational pipeline can read the graph as it is and let structure do the heavy lifting. No domain lore is required to tell the model which interactions to trust or which paths to follow. The same machinery discovers, weights, and composes relations across problems, and it does so consistently. Thus, even when accuracy margins are narrow, the advantage is profound: a pipeline that travels from dataset to dataset, transforming webs of entities into usable signal without custom feature engineering, and solving relational problems end to end in an entirely automatic manner.

Because reusability and automation are central to this claim, we now turn from predictive accuracy to *human effort*. In the next section, we compare the expert data scientist workflow with the relational pipeline, quantifying the marginal hours of human work and the lines of code required to solve a new task under a fixed infrastructure.

[4] Percentages are computed per task as the relative improvement of the better of HTG or HGAT with respect to the baseline, using higher is better for ROC AUC and relative error reduction for MAE.

4.1.1 Human effort

The RelBench[4] study shows that moving from a tabular workflow with manual feature engineering to a relational deep learning approach reduces the effort required to solve any given task.

Following RelBench, we report the *marginal hours of human work*: the number of hours of human work required to solve any new task, excluding all the reusable infrastructure (e.g., data loading logic and training scripts)[4]. In other words, it is the task specific effort a practitioner must invest to take a new task from the raw database to a trained model under a fixed infrastructure, with all reusable components excluded. For the tabular pipeline, in RelBench [4] this is measured as follows: an experienced data scientist executes a five step tabular workflow, namely EDA[5] capped at four hours, one hour of feature ideation, SQL feature engineering with time unconstrained but recorded, model training using a provided LightGBM script with comprehensive hyperparameter tuning that searches up to ten settings, and an optional post hoc inspection; shared tooling is not counted. For the relational pipeline, only the small task specific supervision specification and the schema mapping stored in the training table are counted, since the loader, temporal sampler, trainer, and evaluation scripts are reused among all tasks.

Under this protocol RelBench reports 12.3 ± 1.6 hours per task for the data scientist workflow versus approximately 0.5 hours for RDL, with the RDL solutions consistently under one hour.

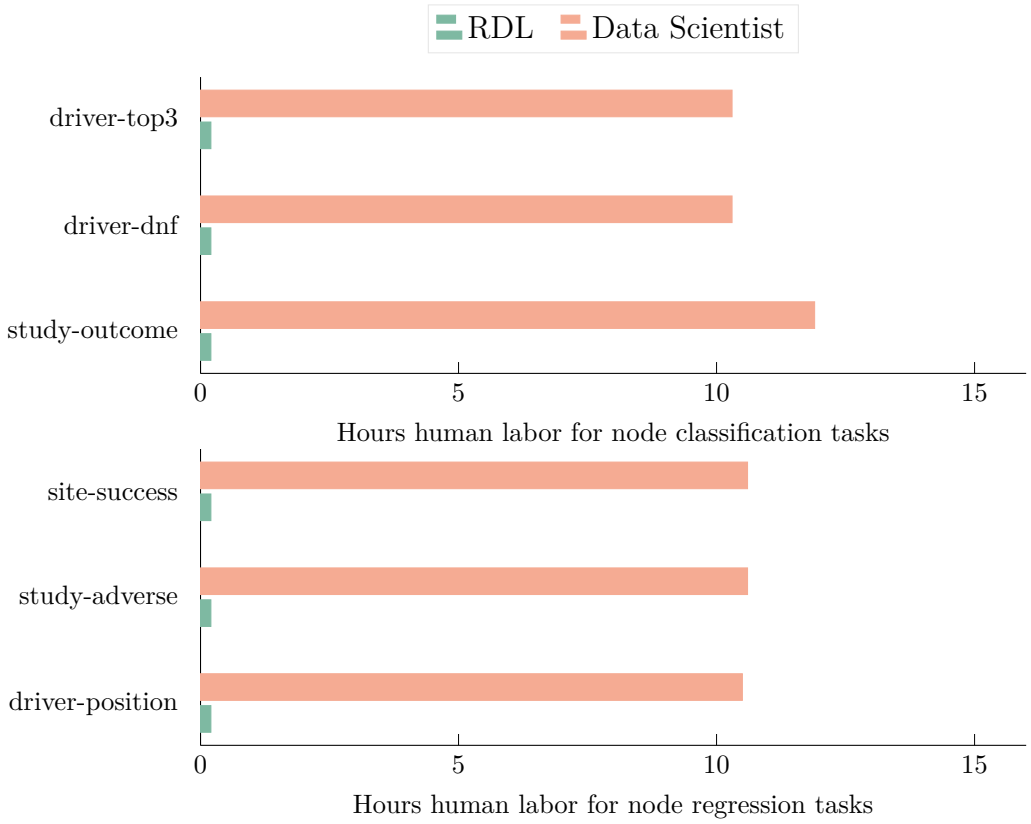
For relational deep learning, we follow the RelBench definition and compute the average number of hours required across our implemented relational models under the same fixed infrastructure, and we observe no significant differences from the results reported by RelBench. This outcome is expected because all our relational backbones reuse the same loader, temporal sampler, trainer, and evaluation scripts described in the RelBench study, while only the supervision specification and the schema mapping are task specific.

Finally, we take the RelBench estimates for the "data scientist" workflow at face value. Independently verifying them would be impractical: it would require recruiting an experienced data scientist for each new task, and carefully controlling for background, tool familiarity, and time allocation; even then, results would depend on practitioner specific choices and heuristics. Different experts would plausibly make different design decisions and achieve different times or scores, undermining comparability across tasks. For these reasons, we adopt the RelBench

[5]EDA stands for Exploratory Data Analysis, the initial phase in which the data scientist quickly explores the dataset and the task to understand the schema, variables, distributions, outliers, and missing values; clarifies the seed time and temporal splits; and produces simple visualizations and checks before any feature engineering or modeling.

figures as the reference point.

Table 4.2: Human work for our tasks under the RelBench user study setting.



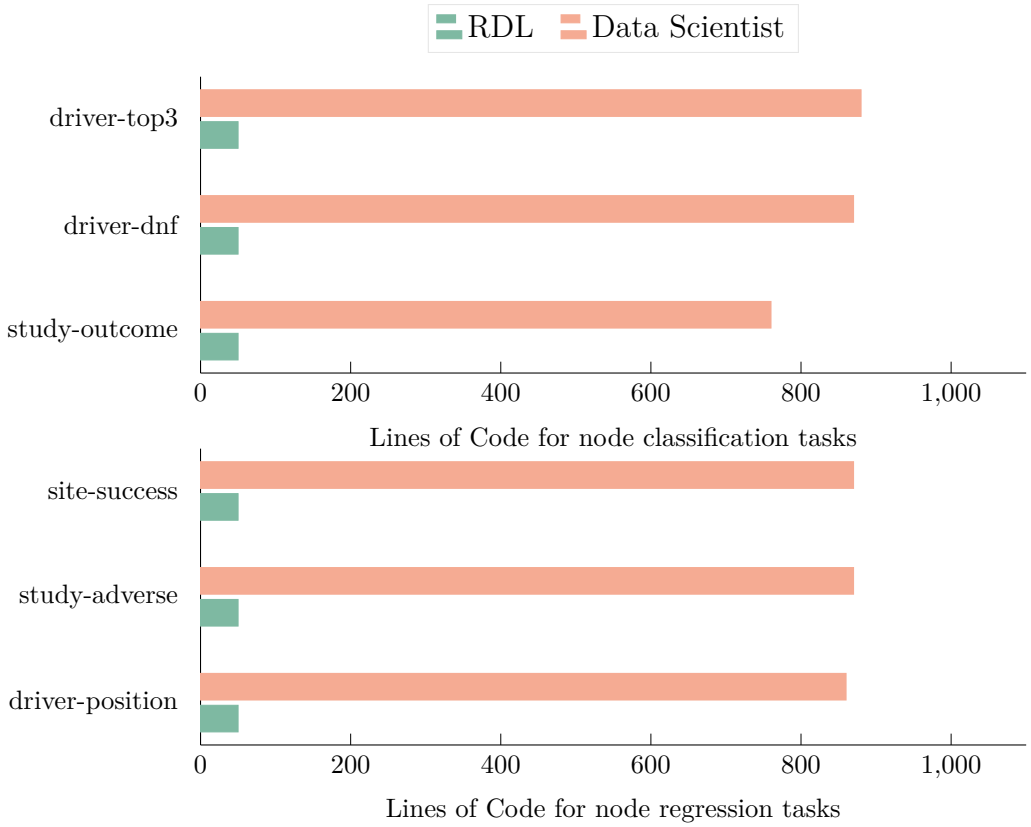
Overall, we observe an average **96%** reduction in the hours of human work for RDL compared to the data scientist workflow.

Relbench[4] also measured the cost in terms of **lines of code** measured as number of lines a practitioner must write to solve a new task, with all reusable components excluded. The tabular workflow counts the EDA and SQL scripts and the manipulations required to feed the provided LightGBM script. For the relational pipeline, only the small specification of training supervision and the schema mapping stored in the training table are counted. Shared infrastructure is excluded in both cases.

Consistent with the previous table, we follow the same protocol for the lines-of-code metric: for RDL, we report the mean number of lines required across our implemented relational models under the fixed infrastructure; for the data scientist workflow, we adopt the figures reported by RelBench without modification.

The figure below reproduces the RelBench comparison for the tasks we study.

Table 4.3: Lines of code for our tasks under the RelBench user study setting.



We can clearly observe that the RDL pipeline requires far fewer lines of code, reflecting its ability to reuse common components and automate the end to end workflow.

Given these findings, one might expect that the additional complexity of the tabular workflow (manual feature ideation and bespoke SQL feature construction) would translate into clear performance gains. Yet the evidence says otherwise: RDL achieves higher test set performance on most tasks, as reported by Robinson et al. [4].

These results show that relational deep learning turns a pipeline into something we can carry from one problem to the next. Once the schema is mapped, the same loader, the same temporal sampler, the same trainer and evaluation can be used again, so the work that usually dissolves at every new task becomes a lasting asset. Reuse is no longer a hope, it is the default: **models change, configurations change, but the pipeline remains.**

At the same time, the relational approach reduces what must be built by hand. There is less bespoke code, fewer fragile transformations, fewer places where leakage

can slip in. The path from raw database to a reliable model grows shorter and cleaner; teams spend their time asking better questions rather than stitching together joins and summaries. Consistency follows from this discipline. Results become easier to compare across tasks and seeds, decisions are documented in configuration rather than scattered across scripts, and the distance between an idea and its test narrows.

In practical terms, the pipeline becomes easier to maintain, easier to audit, and easier to extend. New tasks arrive and are absorbed with small, local choices rather than with weeks of invention. The structure already present in the database carries more of the load, so the system scales with the data rather than with the number of hand crafted features. Less code, fewer surprises, better predictions: this is the quiet advantage that these results make visible.

4.2 Pre-training strategies

We now turn to the results obtained with the pretraining strategies described earlier. Our aim is to understand when pre-training helps, how large the gains are in practice, and where it falls short. We proceed by examining each strategy in turn.

We do not report results for masked attribute prediction (MAP) pre-training because, in our setting, it produced no measurable gains on downstream tasks and remained within the variance across seeds. In our setting, infact, pretraining rests on reconstructing node features, yet many of these features are only weakly relational, often metadata or local signals with little connection to edge dynamics. The reconstruction objective then rewards statistical shortcuts, type averages and co occurrence patterns, rather than forcing the encoder to capture the mechanisms that govern interactions over time. The model optimizes feature fidelity, not structure, and the gain does not transfer to downstream tasks. Put simply, we are searching for relations by reconstructing attributes that do not tell the story of the relations, and the voice of the graph remains in the background.

4.2.1 Edge Dropout

Table 4.4: Effect of EdgeDrop on RelBench tasks. LightGBM and HGraphSAGE numbers are from the RelBench paper[4].

	LightGBM	**HGraphSAGE**		**HTG**		**HGAT**	
		orig.	EdgeDrop	orig.	EdgeDrop	orig.	EdgeDrop
REL F1							
Driver Top 3 ↑ (ROC AUC)	73.92 ± 5.75	**75.54 ± 0.63**	75.48 ± 0.41	75.49 ± 2.91	**76.30 ± 2.05**	78.33 ± 2.14	**80.35 ± 1.81**
Driver Position ↓ (MAE)	4.170 ± 0.137	4.022 ± 0.119	**3.910 ± 0.120**	3.872 ± 0.151	**3.871 ± 0.127**	3.983 ± 0.148	**3.980 ± 0.134**
Driver DNF ↑ (ROC AUC)	68.56 ± 3.89	72.62 ± 0.27	**73.11 ± 0.98**	75.35 ± 1.33	**76.60 ± 1.11**	**73.47 ± 1.07**	72.61 ± 0.36
REL TRIAL							
Study Adverse ↓ (MAE)	44.011 ± 0.998	**44.473 ± 0.209**	44.851 ± 0.391	**47.809 ± 0.417**	50.802 ± 0.271	**48.846 ± 0.302**	50.934 ± 0.289
Study Outcome ↑ (ROC AUC)	70.09 ± 1.41	**68.60 ± 1.01**	68.58 ± 0.65	67.31 ± 2.64	**67.88 ± 2.01**	67.07 ± 1.72	**67.30 ± 0.49**
Site Success ↓ (MAE)	0.425 ± 0.003	0.400 ± 0.020	**0.382 ± 0.011**	0.346 ± 0.091	**0.343 ± 0.072**	0.360 ± 0.052	**0.350 ± 0.055**
REL AVITO							
User Clicks ↑ (ROC AUC)	53.60 ± 0.59	65.90 ± 1.95	**66.68 ± 1.88**	65.98 ± 2.34	**66.19 ± 2.15**	67.23 ± 2.02	**67.98 ± 1.64**
User Visits ↑ (ROC AUC)	53.05 ± 0.32	**66.20 ± 0.10**	66.10 ± 0.14	69.00 ± 1.35	**69.15 ± 1.12**	**66.69 ± 1.61**	66.45 ± 1.20
Ad Ctr ↓ (MAE)	0.041 ± 0.000	**0.041 ± 0.001**	**0.041 ± 0.001**	0.039 ± 0.003	**0.038 ± 0.002**	0.039 ± 0.003	**0.038 ± 0.002**

EdgeDrop is enabled only during training and disabled at validation and test. With $p = 0.3$ on **REL F1**, $p = 0.1$ on **REL TRIAL**, and $p = 0.15$ on **REL AVITO**, it delivers small yet consistent gains: on average, about one percentage point in ROC AUC for classification and ranking, and roughly a one to two percent reduction in MAE for regression. The effect is neutral to positive across GNN backbones, with only occasional small regressions, and it adds no inference overhead and negligible training overhead. While EdgeDrop generally improves robustness by preventing over-reliance on specific edges, we observe a degradation in the Study Adverse (MAE) task. This task appears to rely on localized and informative relations that are partially removed during regularization, reducing the model's ability to capture fine-grained dependencies.

Since EdgeDrop acts primarily as a structural regularization mechanism, large performance shifts are not expected when the model does not exhibit clear signs of overfitting. In our experiments, training and validation scores are already well aligned, suggesting that the base models generalize adequately even without additional regularization. Consequently, EdgeDrop mainly provides small but consistent gains by improving robustness and stability rather than producing major accuracy jumps.

4.2.2 Variational Graph Auto-Encoder (VGAE) pre-training

Table 4.5: Effect of VGAE pre-training on RelBench tasks.

	LightGBM	**HGraphSAGE**		**HTG**		**HGAT**	
		orig.	VGAE	orig.	VGAE	orig.	VGAE
REL F1							
Driver Top 3 ↑ (ROC AUC)	73.92 ± 5.75	75.54 ± 0.63	**75.81 ± 0.85**	75.49 ± 2.91	**76.30 ± 2.15**	78.33 ± 2.14	**80.56 ± 1.98**
Driver Position ↓ (MAE)	4.170 ± 0.137	4.022 ± 0.119	**3.850 ± 0.134**	3.872 ± 0.151	**3.870 ± 0.202**	3.983 ± 0.148	**3.944 ± 0.076**
Driver DNF ↑ (ROC AUC)	68.56 ± 3.89	72.62 ± 0.27	**73.60 ± 0.38**	75.35 ± 1.33	**76.55 ± 1.74**	73.47 ± 1.07	**74.05 ± 0.67**
REL TRIAL							
Study Adverse ↓ (MAE)	44.011 ± 0.998	44.473 ± 0.209	**43.512 ± 0.651**	**47.809 ± 0.417**	48.184 ± 0.626	48.846 ± 0.302	**48.202 ± 0.285**
Study Outcome ↑ (ROC AUC)	70.09 ± 1.41	**68.60 ± 1.01**	68.55 ± 1.51	67.31 ± 2.64	**67.91 ± 3.57**	67.07 ± 1.72	**68.32 ± 1.21**
Site Success ↓ (MAE)	0.425 ± 0.003	0.400 ± 0.020	**0.370 ± 0.022**	0.346 ± 0.091	**0.340 ± 0.061**	0.360 ± 0.052	**0.359 ± 0.074**
REL AVITO							
User Clicks ↑ (ROC AUC)	53.60 ± 0.59	65.90 ± 1.95	**66.75 ± 3.19**	65.98 ± 2.34	**66.35 ± 3.12**	67.23 ± 2.02	**68.05 ± 1.57**
User Visits ↑ (ROC AUC)	53.05 ± 0.32	66.20 ± 0.10	**66.35 ± 0.91**	69.00 ± 1.35	**69.31 ± 2.21**	66.69 ± 1.61	**66.77 ± 0.98**
Ad Ctr ↓ (MAE)	0.041 ± 0.000	**0.041 ± 0.001**	**0.041 ± 0.004**	0.039 ± 0.003	**0.038 ± 0.003**	0.039 ± 0.003	**0.038 ± 0.002**

VGAE pretraining produces modest but generally positive gains, with the largest effects on **REL F1** and on **REL AVITO**. On average, we observe about one percentage point improvement in ROC AUC for classification and roughly one to two percent reduction in MAE for regression, with only modest variability across backbones.

A plausible explanation is that VGAE shapes the encoder toward structural and temporally consistent patterns, improving initialization and robustness, but its link-reconstruction objective is only partially aligned with downstream goals. Gains are therefore larger when relational structure is highly informative or labels are scarce or noisy, and smaller when node features already carry most of the signal. Occasional regressions likely reflect mismatches between the pretraining objective and the task, or backbone-specific sensitivities.

The task appears relatively easy in our setting, as the VGAE pretraining quickly reaches very low reconstruction loss. This suggests a ceiling effect with limited headroom for representation learning. When the encoder already models most edges with high confidence, the pretraining objective provides diminishing returns for downstream tasks. In such cases, modest gains are expected.

4.3 XMetaPath model

We now compare the predictive performance on the test set, placing our black-box architectures alongside the self explainable model (XMetaPath), to assess whether the pursuit of explainability diminishes the model's predictive capacity.

Table 4.6: Comparison of our black-box attention backbones (HTG, HGAT) with the *XMetaPath* self-explainable model. Metrics follow RelBench: ROC AUC is reported with two decimals and MAE with three.

	HTG	**HGAT**	**XMetaPath**
REL F1			
Driver Top 3 ↑ (ROC AUC)	75.49 ± 2.91	78.33 ± 2.14	**82.50 ± 3.31**
Driver Position ↓ (MAE)	3.872 ± 0.151	3.983 ± 0.148	**3.760 ± 0.175**
Driver DNF ↑ (ROC AUC)	75.35 ± 1.33	73.47 ± 1.07	**78.43 ± 1.33**
REL TRIAL			
Study Adverse ↓ (MAE)	47.809 ± 0.417	48.846 ± 0.302	**47.350 ± 0.514**
Study Outcome ↑ (ROC AUC)	**67.31 ± 2.64**	67.07 ± 1.72	**67.31 ± 1.78**
Site Success ↓ (MAE)	**0.346 ± 0.091**	0.360 ± 0.052	0.369 ± 0.055
REL AVITO			
User Clicks ↑ (ROC AUC)	65.98 ± 2.34	**67.23 ± 2.02**	62.81 ± 2.87
User Visits ↑ (ROC AUC)	**69.00 ± 1.35**	66.69 ± 1.61	62.57 ± 2.26
Ad Ctr ↓ (MAE)	0.039 ± 0.003	0.039 ± 0.003	**0.038 ± 0.003**

Prior to analyzing these results, is important to note that in the *rel avito* task, which has the largest number of entities, running the metapath selection with the same hyperparameters used for the other tasks proved computationally onerous, requiring substantial GPU resources and long runtimes. Even Extension 3, which is our default and generally the most efficient variant[6], was affected by these constraints: we could only run a minimal configuration, limiting the number of episodes and enforcing $L_{\max} = 2$. We chose this cap because the bag extension phase was the most computationally demanding component of the entire pipeline.

This curtailed search likely limited the discovery of stronger metapaths. Consequently, performance on this dataset, especially for the classification tasks, does not appear to exceed that of the black box models. In short, the performance gap observed on the REL AVITO dataset is plausibly a consequence of the reduced compute budget rather than an intrinsic limitation of the self-explainable approach.

Regardless of point gains, what matters is that XMetaPath competes with, and often surpasses, black box attention backbones, yet every decision comes with a readable trail. By routing information along a small set of learned meta paths, the model focuses on relations that carry signal and ignores detours that add noise. This selective routing acts like a structural prior that sharpens learning rather than constraining it, which helps explain why accuracy rises while explanations remain

[6]By contrast, Extension 2 was not viable on *rel avito* because the dataset complexity inflated the prompt length beyond the token limits of the free LLM API.

faithful.

The result is a model that we can trust in two senses at once. It performs at the level of strong black box alternatives, and it shows its work. For each prediction we can see which relational routes contributed and by how much, and by aggregating these attributions we obtain a global map of the patterns that matter across the dataset. This makes audits, error analysis, and threshold selection practical, and it turns distribution shifts into visible changes in which paths the model relies on.

Crucially, the metapaths are discovered automatically from the source database. The pipeline explores the relational schema under temporal constraints, proposes candidate relation sequences, and learns which ones improve validation performance. No handcrafted features are required and no domain knowledge is needed to specify which interactions to test. A single well designed pipeline can be applied as is to a new graph, learn the useful metapaths, assign their weights, and deliver both predictions and explanations in a fully automatic and systematic way.

In our implementation, the metapaths used for the reported results come from Extension 3, which employs reinforcement learning to explore the space of relation sequences. An agent walks the graph one relation at a time, receives reward from validation gains and parsimony, and learns to favor the routes that consistently carry signal. This yielded the most meaningful and discriminative meta paths in our setting. Extension 2, the LLM guided search, was robust and extremely fast, but tended to surface less informative metapaths. We attribute this limitation to the constrained context window available in the free tier of the API, which makes it difficult to reason over longer or more intricate relational motifs. In addition, for several tasks this extension was infeasible because the required prompts exceeded the LLM API's token limits. Extension 1 achieved accuracy comparable to Extension 3, but it required considerably more time to decide on the meta paths, since it had to sift through a broad search space with little guidance. In short, reinforcement learning gave us a principled way to turn structure into policy, letting the model discover the routes that matter with both efficiency and clarity.

Chapter 5

Conclusions

This work began with a simple question: if data arrive already relational, why tear apart and discard the relational fabric that gives them meaning?

Is it worth flattening a living web of relations into a silent grid of columns merely to remain within the comfort of tabular models?

Inspired by Robinson et al.[4], we question those assumptions and pursue an alternative that keeps the relational fabric intact and learns directly from it. We build on their work and contribute a practical pipeline, two attention based backbones, an interpretable meta path model, a reinforcement learning procedure for automatic meta path discovery, and an empirical study of self supervised pretraining techniques.

To keep relational structure alive, we need a language that can say who is who, how they connect, and when. The heterogeneous temporal graph is that language. It preserves entity types, states relations explicitly, and anchors every fact to its moment in time. Nothing is flattened, nothing inferred by convenience; keys stay keys, links stay links, time stays ordered. Among data structures it is, arguably, the only one that can model these data without giving up information.

A graph gives us the representation, but we still need a way to act upon it. Graph neural networks provide that way. They define learnable functions on nodes and edges, pass messages along typed relations, and compose evidence across time. Because they operate on the graph directly, they keep types distinct, respect temporal order, and capture both local detail and global context. In practice this turns the database into a machine that learns over structure without handcrafted joins or summaries.

The goal of this thesis has been to explore a spectrum of relational models and design choices, to investigate self supervised pre-training, and to fuse these elements into a robust, reusable pipeline that delivers strong accuracy, together with other desirable qualities, such as systematicity, automation, and, most notably, interpretability.

Within the RelBench protocol [4], we benchmark against the established reference models: a tabular learner, LightGBM, and a relational deep learning backbone, HGraphSAGE. Moving from tabular to relational learning not only reduces manual feature work, it also raises performance whenever the signal is truly relational. Building on these baselines, our self attention models, HTG and HGAT, form a sensible next step. They consistently match or surpass the tabular reference and improve on the relational baseline of Robinson et al.[4], delivering higher accuracy while preserving automation and robustness.

Pretraining contributes a mild but steady lift, not spectacular on any single task yet present across tasks and seeds.

Finally, we wanted a model that does not only predict, it shows its work. With XMetaPath, explanations are native to the computation: each prediction arrives with the meta-paths that carried the signal, their learned weights, and the contribution of the relations within each path. No after the fact tricks, no surrogate rationalizations. And this clarity does not diminish performance; it often helps it. The relational inductive bias concentrates learning on the routes that truly matter; we believe this is why, in our experiments, accuracy rises as the explanations sharpen. In short, a system built for results and for explainability, where accuracy and understanding travel together.

Crucially, the meta paths that support these explanations are not handcrafted; they are discovered automatically from the source database. In our experiments the most meaningful and discriminative routes come from Extension 3, which uses reinforcement learning to walk relation sequences under temporal constraints and rewards those that lift validation while requiring fewer short training runs to obtain validation signals. Extension 2, the LLM guided search, proved robust yet tended to surface less informative routes, a limitation we attribute to small prompt window of the free API. Extension 1 reached comparable accuracy but requires too much time to select routes, since it sifted a very broad search space with little guidance; its cost grows sharply with schema size and number of relations, making it impractical for schemas with many tables.

There are limits and there is room to grow. First, we can explore cross database pre training, where a model is trained on a large source schema and then fine tuned on each downstream graph. This may yield stronger invariances, better sample efficiency, and more stable transfer across domains. Second, we can make meta path selection faster and more efficient, replacing heavy searches with light weight policies. Extension 2 was designed precisely for this goal, replacing heavy searches with fast and parallelizable LLM calls. Future work should test it with LLMs that offer a larger context window and richer prompts. Third, in the current XMetaPath the parameter X that controls the number of meta paths is fixed a priori and chosen before execution; yet the appropriate value should be task dependent and data dependent, and it is not easy to tune reliably. Finally, we

can leverage large language models to produce high level predictions and human friendly explanations by verbalizing the selected meta paths and by providing a few representative node instances as structured context for the language model. This follows recent work that uses verbalized metapaths as contextual augmentation for LLMs[51], which we can adapt to our domain, turning the selected meta paths, their learned weights, and a few exemplar neighbors into compact narratives that reveal where the model looked, why those relations mattered, and how each strand of evidence nudged the prediction, offering the end user a clear, faithful, and human readable view of the reasoning, along with gentle what if reflections on what could have shifted the outcome.

Relational databases are not just everywhere, they are the quiet infrastructure beneath our daily decisions. They store facts with care, relate entities with purpose, and speak a language, SQL, that lets us ask precise questions across time and tables. In this medium, data carry two kinds of meaning at once: local attributes that describe each record, and structural constraints that bind records together. It is a world that is both **granular and connected**, **immediate and relational**. Whoever learns to use both facets at once can turn ubiquity into advantage.

Relational modeling is not only about surpassing tabular baselines; it is about making relational reasoning automatic and systematic, about having a **consistent answers for all problems**. A single well designed pipeline can be applied unchanged to a new graph, learn which routes matter, assign their weights, and deliver predictions with reasons, without domain knowledge and without bespoke feature engineering.

This is the promise we explore: to replace manual, domain heavy flattening with a pipeline that reads the graph as it is, learns from it directly, and returns predictions with reasons. We compare strong tabular baselines with relational backbones, study self supervised pretraining, and introduce XMetaPath, a self-explainable model that learns which routes matter and shows its work. The goal is to build an automatic and systematic pipeline that delivers strong predictive performance while keeping the reasoning transparent by design and directly accessible, so the model exposes its logic and provides faithful local and global explanations, ensuring that every prediction arrives with a traceable rationale and competitive test scores.

We end where we began. A database is more than a store of records; it is a living weave of facts and relations, of detail and shape. When learning respects both, it does more than predict: it brings causal structure into focus. Temporal order and relational constraints become visible, revealing patterns a flat table would otherwise hide. In that fidelity we find not only stronger predictions, but also the shape of cause and effect coming into focus.

Bibliography

[1] Kristi Berg, Dr. Tom Seymour, and Richa Goel. «History Of Databases». In: *International Journal of Management Information Systems (IJMIS)* 17 (Dec. 2012), p. 29. DOI: 10.19030/ijmis.v17i1.7587 (cit. on p. 1).

[2] Alistair E W Johnson et al. «MIMIC-III, a freely accessible critical care database». In: *Scientific data* 3.1 (2016), pp. 1–9 (cit. on p. 1).

[3] Vijay Prakash Dwivedi, Charilaos Kanatsoulis, Shenyang Huang, and Jure Leskovec. *Relational Deep Learning: Challenges, Foundations and Next-Generation Architectures*. 2025. arXiv: 2506.16654 [cs.LG]. URL: https://arxiv.org/abs/2506.16654 (cit. on p. 1).

[4] Joshua Robinson et al. *RelBench: A Benchmark for Deep Learning on Relational Databases*. 2024. arXiv: 2407.20060 [cs.LG]. URL: https://arxiv.org/abs/2407.20060 (cit. on pp. 1, 2, 4, 5, 7, 47, 58, 79, 106, 121–123, 125–127, 129, 133, 134).

[5] Tianqi Chen and Carlos Guestrin. «XGBoost: A Scalable Tree Boosting System». In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. ACM, Aug. 2016, pp. 785–794. DOI: 10.1145/2939672.2939785. URL: http://dx.doi.org/10.1145/2939672.2939785 (cit. on pp. 2, 8, 9).

[6] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. «LightGBM: A Highly Efficient Gradient Boosting Decision Tree». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf (cit. on p. 2).

[7] Veronica Lachi, Antonio Longa, Beatrice Bevilacqua, Bruno Lepri, Andrea Passerini, and Bruno Ribeiro. «Boosting Relational Deep Learning with Pretrained Tabular Models». In: *arXiv preprint arXiv:2504.04934* (2025) (cit. on p. 2).

[8] Guozhu Dong and Huan Liu. *Feature Engineering for Machine Learning and Data Analytics*. CRC Press, 2018 (cit. on p. 2).

[9] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. «Position: Relational Deep Learning - Graph Representation Learning on Relational Databases». In: *Proceedings of the 41st International Conference on Machine Learning*. Ed. by Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp. Vol. 235. Proceedings of Machine Learning Research. PMLR, 21–27 Jul 2024, pp. 13592–13607. URL: https://proceedings.mlr.press/v235/fey24a.html (cit. on pp. 4, 5, 47, 58).

[10] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. «The Graph Neural Network Model». In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605 (cit. on pp. 4, 11, 23).

[11] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. «Neural message passing for quantum chemistry». In: *International conference on machine learning*. PMLR. 2017, pp. 1263–1272 (cit. on pp. 4, 21).

[12] Guolin Ke, Qiqi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. «LightGBM: A Highly Efficient Gradient Boosting Decision Tree». In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf (cit. on pp. 6, 8, 123).

[13] Tianqi Chen and Carlos Guestrin. «XGBoost: A Scalable Tree Boosting System». In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016, pp. 785–794 (cit. on pp. 6, 9).

[14] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. «Heterogeneous graph attention network». In: *Proceedings of the World Wide Web Conference*. 2019, pp. 2022–2032 (cit. on p. 6).

[15] Jan Peleska and Zdenek Šır. «Transformers Meet Relational Databases: Self-Attention for Joint Encoding of Tables and Schemas». In: *arXiv preprint arXiv:2403.01583* (2024) (cit. on p. 6).

[16] Pratt Institute. *Visualizing Les Misérables Social Network (1862)*. Accessed: 2025-10-03. n.d. URL: https://studentwork.prattsi.org/infovis/visualization/visualizing-les-miserables-social-network-1862/ (cit. on p. 11).

[17] Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao. *Graph Neural Networks: Foundations, Frontiers, and Applications*. Singapore: Springer Singapore, 2022, p. 725 (cit. on p. 11).

[18] William L. Hamilton. «Graph Representation Learning». In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14.3 (), pp. 1–159 (cit. on pp. 12–14, 23).

[19] Ruiqi Li, Peishun Jiao, and Junyi Li. «PF2PI: Protein Function Prediction Based on AlphaFold2 Information and Protein-Protein Interaction». In: *Advanced Intelligent Computing in Bioinformatics*. Ed. by De-Shuang Huang, Yijie Pan, and Qinhu Zhang. Singapore: Springer Nature Singapore, 2024, pp. 278–289. ISBN: 978-981-97-5692-6 (cit. on p. 12).

[20] Yang Li, Kangbo Liu, Ranjan Satapathy, Suhang Wang, and Erik Cambria. *Recent Developments in Recommender Systems: A Survey*. 2023. arXiv: 2306.12680 [cs.IR]. URL: https://arxiv.org/abs/2306.12680 (cit. on p. 15).

[21] Robin Burke, Alexander Felfernig, and Mehmet Göker. «Recommender Systems: An Overview». In: *Ai Magazine* 32 (Sept. 2011), pp. 13–18. DOI: 10.1609/aimag.v32i3.2361 (cit. on p. 15).

[22] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. «Translating Embeddings for Modeling Multi-relational Data». In: *NIPS*. 2013 (cit. on p. 15).

[23] Qimai Li, Zhichao Han, and Xiao-Ming Wu. *Deeper Insights into Graph Convolutional Networks for Semi-Supervised Learning*. 2018. arXiv: 1801.07606 [cs.LG]. URL: https://arxiv.org/abs/1801.07606 (cit. on p. 22).

[24] Kenta Oono and Taiji Suzuki. *Graph Neural Networks Exponentially Lose Expressive Power for Node Classification*. 2021. arXiv: 1905.10947 [cs.LG]. URL: https://arxiv.org/abs/1905.10947 (cit. on p. 22).

[25] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2018. arXiv: 1706.02216 [cs.SI]. URL: https://arxiv.org/abs/1706.02216 (cit. on pp. 25, 53, 56, 58).

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 26, 28, 29, 58, 60, 64).

[27] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. «Graph attention networks». In: *International Conference on Learning Representations (ICLR)*. 2018 (cit. on pp. 26, 27, 29, 32, 57, 58).

[28] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805 (cit. on p. 29).

[29] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL]. URL: https://arxiv.org/abs/1409.0473 (cit. on p. 29).

[30] Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. *Modeling Relational Data with Graph Convolutional Networks*. 2017. arXiv: 1703.06103 [stat.ML]. URL: https://arxiv.org/abs/1703.06103 (cit. on p. 29).

[31] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. 2016. arXiv: 1502.03044 [cs.LG]. URL: https://arxiv.org/abs/1502.03044 (cit. on p. 29).

[32] Chengxuan 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?* 2021. arXiv: 2106.05234 [cs.LG]. URL: https://arxiv.org/abs/2106.05234 (cit. on pp. 29, 58, 60, 64).

[33] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Peng Cui, P. Yu, and Yanfang Ye. *Heterogeneous Graph Attention Network*. 2021. arXiv: 1903.07293 [cs.SI]. URL: https://arxiv.org/abs/1903.07293 (cit. on pp. 30, 37, 38, 58).

[34] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. *Graph Neural Networks for Social Recommendation*. 2019. arXiv: 1902.07243 [cs.IR]. URL: https://arxiv.org/abs/1902.07243 (cit. on p. 34).

[35] «Graph Convolutional Networks with Markov Random Field Reasoning for Social Spammer Detection». In: 34 (Apr. 2020), pp. 1054–1061. DOI: 10.1609/aaai.v34i01.5455. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5455 (cit. on p. 34).

[36] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. «Graph Convolutional Neural Networks for Web-Scale Recommender Systems». In: *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*. KDD '18. ACM, July 2018, pp. 974–983. DOI: 10.1145/3219819.3219890. URL: http://dx.doi.org/10.1145/3219819.3219890 (cit. on p. 34).

[37] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. *Interaction Networks for Learning about Objects, Relations and Physics*. 2016. arXiv: 1612.00222 [cs.AI]. URL: https://arxiv.org/abs/1612.00222 (cit. on p. 34).

[38] Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. «Knowledge Base Completion with Out-of-Knowledge-Base Entities: A Graph Neural Network Approach». In: *Transactions of the Japanese Society for Artificial Intelligence* 33.2 (2018), F-H72$_1^{\smile}$10. ISSN: 1346-8030. DOI: 10.1527/tjsai.f-h72. URL: http://dx.doi.org/10.1527/tjsai.F-H72 (cit. on p. 34).

[39] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. *Neural Message Passing for Quantum Chemistry*. 2017. arXiv: 1704.01212 [cs.LG]. URL: https://arxiv.org/abs/1704.01212 (cit. on p. 34).

[40] Jaykumar Kakkad, Jaspal Jannu, Kartik Sharma, Charu Aggarwal, and Sourav Medya. *A Survey on Explainability of Graph Neural Networks*. 2023. arXiv: 2306.01958 [cs.LG]. URL: https://arxiv.org/abs/2306.01958 (cit. on p. 35).

[41] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. «GNNExplainer: Generating explanations for graph neural networks». In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019 (cit. on p. 36).

[42] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. *Parameterized Explainer for Graph Neural Network*. 2020. arXiv: 2011.04573 [cs.LG]. URL: https://arxiv.org/abs/2011.04573 (cit. on p. 36).

[43] Yaomin Chang, Chuan Chen, Weibo Hu, Zibin Zheng, Xiaocong Zhou, and Shouzhi Chen. «Megnn: Meta-path extracted graph neural network for heterogeneous graph representation learning». In: *Knowledge-Based Systems* 235 (2022), p. 107611. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2021.107611. URL: https://www.sciencedirect.com/science/article/pii/S095070512100873X (cit. on p. 38).

[44] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. «MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding». In: *Proceedings of The Web Conference 2020*. WWW '20. ACM, Apr. 2020, pp. 2331–2341. DOI: 10.1145/3366423.3380297. URL: http://dx.doi.org/10.1145/3366423.3380297 (cit. on p. 38).

[45] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. *Heterogeneous Graph Transformer*. 2020. arXiv: 2003.01332 [cs.LG]. URL: https://arxiv.org/abs/2003.01332 (cit. on p. 38).

[46] Qingsong Lv et al. *Are we really making much progress? Revisiting, benchmarking, and refining heterogeneous graph neural networks*. 2021. arXiv: 2112.14936 [cs.LG]. URL: https://arxiv.org/abs/2112.14936 (cit. on p. 39).

[47] Seongjun Yun, Minbyul Jeong, Sungdong Yoo, Seunghun Lee, Sean S. Yi, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. *Graph Transformer Networks: Learning Meta-path Graphs to Improve GNNs*. 2021. arXiv: 2106.06218 [cs.LG]. URL: https://arxiv.org/abs/2106.06218 (cit. on pp. 39, 46).

[48] Francesco Ferrini, Antonio Longa, Andrea Passerini, and Manfred Jaeger. *Meta-Path Learning for Multi-relational Graph Neural Networks*. 2023. arXiv: 2309.17113 [cs.LG]. URL: https://arxiv.org/abs/2309.17113 (cit. on pp. 39, 91).

[49] Anasua Mitra, Priyesh Vijayan, Sanasam Ranbir Singh, Diganta Goswami, Srinivasan Parthasarathy, and Balaraman Ravindran. «Revisiting Link Prediction on Heterogeneous Graphs with a Multi-view Perspective». In: *2022 IEEE International Conference on Data Mining (ICDM)*. 2022, pp. 358–367. DOI: 10.1109/ICDM54844.2022.00046 (cit. on p. 39).

[50] Francesco Ferrini, Antonio Longa, Andrea Passerini, and Manfred Jaeger. *A Self-Explainable Heterogeneous GNN for Relational Deep Learning*. 2025. arXiv: 2412.00521 [cs.LG]. URL: https://arxiv.org/abs/2412.00521 (cit. on pp. 39, 44, 91, 98, 100–102).

[51] Harshvardhan Solanki, Jyoti Singh, Yihui Chong, and Ankur Teredesai. «Metapath of thoughts: Verbalized metapaths in heterogeneous graph as contextual augmentation to LLM». In: (2024). URL: https://www.amazon.science/publications/metapath-of-thoughts-verbalized-metapaths-in-heterogeneous-graph-as-contextual-augmentation-to-llm (cit. on pp. 46, 104, 135).

[52] Weihua Hu, Yiwen Yuan, Zecheng Zhang, Akihiro Nitta, Kaidi Cao, Vid Kocijan, Jinu Sunil, Jure Leskovec, and Matthias Fey. *PyTorch Frame: A Modular Framework for Multi-Modal Tabular Learning*. 2024. arXiv: 2404.00776 [cs.LG]. URL: https://arxiv.org/abs/2404.00776 (cit. on p. 48).

[53] Lun Li, David Alderson, Reiko Tanaka, John C. Doyle, and Walter Willinger. *Towards a Theory of Scale-Free Graphs: Definition, Properties, and Implications (Extended Version)*. 2005. arXiv: cond-mat/0501169 [cond-mat.dis-nn]. URL: https://arxiv.org/abs/cond-mat/0501169 (cit. on p. 60).

[54] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. *A Simple Framework for Contrastive Learning of Visual Representations*. 2020. arXiv: 2002.05709 [cs.LG]. URL: https://arxiv.org/abs/2002.05709 (cit. on p. 72).

[55] Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. *wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations*. 2020. arXiv: 2006.11477 [cs.CL]. URL: https://arxiv.org/abs/2006.11477 (cit. on p. 72).

[56] Alec Radford et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021. arXiv: 2103.00020 [cs.CV]. URL: https://arxiv.org/abs/2103.00020 (cit. on p. 72).

[57] Beatrice Bevilacqua, Joshua Robinson, Jure Leskovec, and Bruno Ribeiro. «Holographic Node Representations: Pre-training Task-Agnostic Node Embeddings». In: *The Thirteenth International Conference on Learning Representations*. 2025. URL: https://openreview.net/forum?id=tGYFikNONB (cit. on p. 72).

[58] Quang Truong, Zhikai Chen, Mingxuan Ju, Tong Zhao, Neil Shah, and Jiliang Tang. *A Pre-training Framework for Relational Data with Information-theoretic Principles*. 2025. arXiv: 2507.09837 [cs.LG]. URL: https://arxiv.org/abs/2507.09837 (cit. on p. 72).

[59] Jun Xia, Yanqiao Zhu, Yuanqi Du, and Stan Z. Li. *A Survey of Pretraining on Graphs: Taxonomy, Methods, and Applications*. 2022. arXiv: 2202.07893 [cs.LG]. URL: https://arxiv.org/abs/2202.07893 (cit. on pp. 72, 73).

[60] Jiezhong Qiu, Qibin Chen, Yuxiao Dong, Jing Zhang, Hongxia Yang, Ming Ding, Kuansan Wang, and Jie Tang. «GCC: Graph Contrastive Coding for Graph Neural Network Pre-Training». In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*. KDD '20. ACM, Aug. 2020, pp. 1150–1160. DOI: 10.1145/3394486.3403168. URL: http://dx.doi.org/10.1145/3394486.3403168 (cit. on p. 73).

[61] Zhenyu Hou, Xiao Liu, Yukuo Cen, Yuxiao Dong, Hongxia Yang, Chunjie Wang, and Jie Tang. «GraphMAE: Self-Supervised Masked Graph Autoencoders». In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Also available as arXiv:2205.10803. Association for Computing Machinery, 2022. DOI: 10.1145/3534678.3539321. URL: https://doi.org/10.1145/3534678.3539321 (cit. on p. 73).

[62] Zhenyu Hou, Yufei He, Yukuo Cen, Xiao Liu, Yuxiao Dong, Evgeny Kharlamov, and Jie Tang. «GraphMAE2: A Decoding-Enhanced Masked Self-Supervised Graph Learner». In: *Proceedings of the ACM Web Conference 2023*. Also available as arXiv:2304.04779. Association for Computing Machinery, 2023. DOI: 10.1145/3543507.3583379. URL: https://doi.org/10.1145/3543507.3583379 (cit. on p. 73).

[63] Yijun Tian, Kaiwen Dong, Chunhui Zhang, Chuxu Zhang, and Nitesh V. Chawla. «Heterogeneous Graph Masked Autoencoders». In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 8. Also available as arXiv:2208.09957. AAAI Press, 2023. URL: https://ojs.aaai.org/index.php/AAAI/article/view/26192 (cit. on p. 73).

[64] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. «Strategies for Pre-training Graph Neural Networks». In: *International Conference on Learning Representations*. Spotlight; also available as arXiv:1905.12265. 2020. URL: https://openreview.net/forum?id=HJlWWJSFDH (cit. on p. 73).

[65] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. «DropEdge: Towards Deep Graph Convolutional Networks on Node Classification». In: *International Conference on Learning Representations*. Also available as arXiv:1907.10903. 2020. URL: https://openreview.net/forum?id=Hkx1qkrKPr (cit. on p. 73).

[66] Thomas N. Kipf and Max Welling. *Variational Graph Auto-Encoders*. 2016. arXiv: 1611.07308 [stat.ML]. URL: https://arxiv.org/abs/1611.07308 (cit. on pp. 79, 85).

[67] IBM Think. *Variational Autoencoder (VAE)*. Accessed: 2025-08-20. n.d. URL: https://www.ibm.com/it-it/think/topics/variational-autoencoder (cit. on p. 80).

[68] Abhishek Dalvi, Ayan Acharya, Jing Gao, and Vasant G. Honavar. «Variational Graph Auto-Encoders for Heterogeneous Information Network». In: *NeurIPS 2022 Workshop on New Frontiers in Graph Learning (GLFrontiers)*. Poster. 2022. URL: https://openreview.net/forum?id=-l2yynwJWtX (cit. on p. 83).

[69] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. «DropEdge: Towards Deep Graph Convolutional Networks on Node Classification». In: *arXiv preprint arXiv:1907.10903* (2019). URL: https://arxiv.org/abs/1907.10903 (cit. on p. 88).

[70] Antonia Creswell, Murray Shanahan, and Irina Higgins. *Selection-Inference: Exploiting Large Language Models for Interpretable Logical Reasoning*. 2022. arXiv: 2205.09712 [cs.AI]. URL: https://arxiv.org/abs/2205.09712 (cit. on p. 104).

[71] Hossein Bahak, Farzaneh Taheri, Zahra Zojaji, and Arefeh Kazemi. *Evaluating ChatGPT as a Question Answering System: A Comprehensive Analysis and Comparison with Existing Models*. 2023. arXiv: 2312.07592 [cs.CL]. URL: https://arxiv.org/abs/2312.07592 (cit. on p. 104).

[72] Tianyi Zhang, Faisal Ladhak, Esin Durmus, Percy Liang, Kathleen McKeown, and Tatsunori B. Hashimoto. *Benchmarking Large Language Models for News Summarization*. 2023. arXiv: 2301.13848 [cs.CL]. URL: https://arxiv.org/abs/2301.13848 (cit. on p. 104).

[73] Arkadeep Acharya, Brijraj Singh, and Naoyuki Onoe. «LLM Based Generation of Item-Description for Recommendation System». In: *Proceedings of the 17th ACM Conference on Recommender Systems*. RecSys '23. Singapore, Singapore: Association for Computing Machinery, 2023, pp. 1204–1207. ISBN: 9798400702419. DOI: 10.1145/3604915.3610647. URL: https://doi.org/10.1145/3604915.3610647 (cit. on p. 104).

[74] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155 (cit. on p. 104).

[75] Marek Wydmuch, Łukasz Borchmann, and Filip Graliński. *Tackling prediction tasks in relational databases with LLMs*. 2024. arXiv: 2411.11829 [cs.LG]. URL: https://arxiv.org/abs/2411.11829 (cit. on pp. 104–106).

[76] Julian Schrittwieser et al. «Mastering Atari, Go, chess and shogi by planning with a learned model». In: *Nature* 588.7839 (Dec. 2020), pp. 604–609. ISSN: 1476-4687. DOI: 10.1038/s41586-020-03051-4. URL: http://dx.doi.org/10.1038/s41586-020-03051-4 (cit. on p. 111).

[77] Dmitry Kalashnikov et al. *QT-Opt: Scalable Deep Reinforcement Learning for Vision-Based Robotic Manipulation*. 2018. arXiv: 1806.10293 [cs.LG]. URL: https://arxiv.org/abs/1806.10293 (cit. on p. 111).

[78] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. «Deep Reinforcement Learning from Human Preferences». In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc., 2017. URL: https://proceedings.neurips.cc/paper_files/paper/2017/file/d5e2c0adad503c91f91df240d0cd4e49-Paper.pdf (cit. on p. 111).

[79] Wentao Ning et al. «Automatic Meta-Path Discovery for Effective Graph-Based Recommendation». In: *Proceedings of the 31st ACM International Conference on Information amp; Knowledge Management*. CIKM '22. ACM, Oct. 2022, pp. 1563–1572. DOI: 10.1145/3511808.3557244. URL: http://dx.doi.org/10.1145/3511808.3557244 (cit. on p. 111).

[80] Guojia Wan, Bo Du, Shirui Pan, and Gholameza Haffari. «Reinforcement Learning Based Meta-Path Discovery in Large-Scale Heterogeneous Information Networks». In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (Apr. 2020), pp. 6094–6101. DOI: 10.1609/aaai.v34i04.6073. URL: https://ojs.aaai.org/index.php/AAAI/article/view/6073 (cit. on p. 111).

[81] Wentao Ning et al. *Automatic Meta-Path Discovery for Effective Graph-Based Recommendation*. 2022. DOI: https://doi.org/10.1145/3511808.3557244. arXiv: 2112.12845 [cs.IR]. URL: https://arxiv.org/abs/2112.12845 (cit. on p. 111).

[82] Shaina Raza, Mizanur Rahman, Safiullah Kamawal, Armin Toroghi, Ananya Raval, Farshad Navah, and Amirmohammad Kazemeini. *A Comprehensive Review of Recommender Systems: Transitioning from Theory to Practice*. 2025. arXiv: 2407.13699 [cs.IR]. URL: https://arxiv.org/abs/2407.13699.

[83] Veronica Lachi, Antonio Longa, Beatrice Bevilacqua, Bruno Lepri, Andrea Passerini, and Bruno Ribeiro. *Boosting Relational Deep Learning with Pretrained Tabular Models*. 2025. arXiv: 2504.04934 [cs.DB]. URL: https://arxiv.org/abs/2504.04934.

[84] Thomas N Kipf and Max Welling. «Semi-supervised classification with graph convolutional networks». In: *International Conference on Learning Representations (ICLR)*. 2017.

[85] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG]. URL: https://arxiv.org/abs/1609.02907.

[86] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. «MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding». In: *Proceedings of The Web Conference 2020*. WWW '20. ACM, Apr. 2020, pp. 2331–2341. DOI: 10.1145/3366423.3380297. URL: http://dx.doi.org/10.1145/3366423.3380297.

[87] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. «Graph neural networks: A review of methods and applications». In: *AI Open* 1 (2020), pp. 57–81.

[88] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. «A comprehensive survey on graph neural networks». In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2020), pp. 4–24.

[89] Somdutta Ganguli and Shivam Thakur. «Machine learning based recommendation system». In: *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. IEEE. 2020, pp. 660–664.

[90] DB-Engines. *DB-Engines Ranking – August 2025*. Online; accessed 7 Aug 2025. https://db-engines.com/en/ranking. 2025.

[91] Gyubok Lee, Sunjun Kweon, Seongsu Bae, and Edward Choi. «Overview of the EHRSQL 2024 Shared Task on Reliable Text-to-SQL Modeling on Electronic Health Records». In: *Proceedings of the 6th Clinical Natural Language Processing Workshop*. Association for Computational Linguistics. 2024, pp. 644–654. URL: https://aclanthology.org/2024.clinicalnlp-1.62.

[92] Sergiu-Alexandru Ionescu, Vlad Diaconita, and Andreea-Oana Radu. «Engineering Sustainable Data Architectures for Modern Financial Institutions». In: *Electronics* 14.8 (2025), p. 1650. DOI: 10.3390/electronics14081650.

[93] Jannette G. Zapata. «MySQL vs PostgreSQL: A Comparative Analysis of RDBMS Response Time in Web-based E-commerce». In: *International Journal of Computing Studies* (2025). Online: https://doi.org/10.13140/RG.2.2.24791.69288.

[94] ATLAS Collaboration. «Software and Computing for Run 3 of the ATLAS Experiment at the LHC». In: *European Physical Journal C* 85 (2025), p. 234. DOI: 10.1140/epjc/s10052-024-13701-w.

[95] Ruslan Mashinistov, Lino Gerlach, Paul Laycock, Andrea Formica, Giacomo Govi, and Chris Pinkenburg. «The HSF Conditions Database Reference Implementation». In: *Proceedings of CHEP 2024*. Vol. 295. EPJ Web of Conferences. 2024, p. 01051. DOI: 10.1051/epjconf/202429501051.

[96] Wisal Khan, Teerath Kumar, Cheng Zhang, Kislay Raj, Arunabha M. Roy, and Bin Luo. «SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic Literature Review». In: *Big Data and Cognitive Computing* 7.2 (2023), p. 97. DOI: 10.3390/bdcc7020097.

[97] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, and Marco Tulio Ribeiro. *ART: Automatic multi-step reasoning and tool-use for large language models*. 2023. arXiv: 2303.09014 [cs.CL]. URL: https://arxiv.org/abs/2303.09014.

[98] Yige Zhao, Jianxiang Yu, Yao Cheng, Chengcheng Yu, Yiding Liu, Xiang Li, and Shuaiqiang Wang. *Variational Graph Autoencoder for Heterogeneous Information Networks with Missing and Inaccurate Attributes*. 2023. arXiv: 2311.07929 [cs.LG]. URL: https://arxiv.org/abs/2311.07929.

[99] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. *Strategies for Pre-training Graph Neural Networks*. 2020. arXiv: 1905.12265 [cs.LG]. URL: https://arxiv.org/abs/1905.12265.

[100] Xunqiang Jiang, Tianrui Jia, Yuan Fang, Chuan Shi, Zhe Lin, and Hui Wang. «Pre-training on Large-Scale Heterogeneous Graph». In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. Virtual Event, Singapore: Association for Computing Machinery, 2021, pp. 756–766. ISBN: 9781450383325. DOI: 10.1145/3447548.3467396. URL: https://doi.org/10.1145/3447548.3467396.

[101] Difan Luo, Wei Cheng, Wenchao Yu, Bo Zong, Jingchao Ni, Haifeng Chen, and Xiang Zhang. «Parameterized explainer for graph neural network». In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 19620–19631.

[102] Weihua Hu and Matthias Fey. *TorchFrame: A Unified Library for (Relational) Tabular Data*. GitHub repository. 2024. URL: https://github.com/pyg-team/torch-frame.