Politecnico di Torino

Master's Degree in Physics of Complex Systems

Master's Degree Thesis

Static Representation of Temporal Graphs

Supervisors:

Prof. Andrea Pagnani
Dr. Claudio Borile
Dr. André Panisson

Candidate:

Silvia Di Giovanni

Academic Year 2024-2025

Abstract

Graph Neural Networks (GNNs) have proven to be a powerful tool for machine learning tasks on static graphs, such as node classification, link prediction, and graph classification. However, many real-world networks are dynamic in nature, with edges and node attributes changing over time. Fully dynamic GNN architectures successfully capture temporal patterns, but they often come with high computational costs, complex training pipelines, and limited scalability.

This thesis investigates an alternative approach: mapping dynamic graphs into an equivalent static representation, enabling the use of simpler and more efficient GNN models while still preserving essential temporal information. We study the problem in the context of supervised dynamic link prediction, introducing two dynamic-to-static graph mapping strategies and several GNN architectures designed to operate on the resulting representations.

Our experiments highlight the benefits and limitations of this approach. We show that lightweight, static models can achieve competitive performance on dynamic link prediction tasks when supported by carefully constructed temporal graph snapshots. At the same time, we identify key challenges that limit the expressiveness of purely static representations, including embedding scarcity, cold-start issues, and the difficulty of capturing long-range temporal dependencies.

Overall, this work positions dynamic-to-static graph mappings as a practical middle ground between static GNNs and fully dynamic architectures. The proposed framework offers a scalable and modular alternative for learning on temporal networks, while also opening several opportunities for future development in temporal encoding and hybrid dynamic architectures.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many aspects of our lives can be naturally represented using graphs. Social networks, recommendation systems, mobility patterns, power grids, and even physical experiments all exhibit underlying relational structures that can be modeled as networks. While early graph analysis relied on classical algorithms and predefined structural measures, more recent work has shifted toward graph representation learning approaches, beginning with the pioneering Graph Convolutional Networks (GCNs) [25], which introduced an expressive, learnable framework for node representation learning.

Over the years, increasingly expressive architectures have been proposed, beginning with the earliest GCNs [25], progressing to attention-based approaches such as GAT [50], and extending to inductive frameworks like GraphSAGE [14]. These models are built on the principle of message aggregation from neighboring nodes, enabling effective learning on both homogeneous and heterogeneous graph structures. The resulting SOTA models span a wide range of applications and support a rapidly growing research area that is increasingly essential in modern data-driven systems. Nevertheless, static graph representations overlook important aspects that only a fully dynamic formulation, where information changes over time, can capture. The temporal behavior of a system is often crucial, revealing patterns and dependencies that would remain hidden if one relied exclusively on a fixed, time-independent version of the data.

From this need emerged the study of temporal graphs, along with a variety of methods developed to analyze their structure and dynamics. Early approaches built upon preexisting static graph techniques: temporal data was converted into a series of fixed snapshots on which traditional GNNs were applied, at the cost of losing temporal information due to fragmentation. The following step was the introduction of discrete-time models (e.g. EvolveGCN [35], DySAT [41]) that explicitly encode the evolving nature of

the graph across successive snapshots. In parallel, continuous-time models such as Temporal Graph Networks (TGN) [39], TGAT [52], and DyRep [49] were developed, representing events as lists of temporally ordered interactions. These approaches enable temporal graph analysis to fully exploit the continuous evolution of the system, capturing dynamics that snapshot-based methods might miss and providing a richer foundation for subsequent studies.

However, both lines of study have limitations. Continuous-time approaches often require complex architectures and operate on fine-grained temporal information, resulting in high computational costs and scalability challenges if applied on large datasets. Discrete-time models, while being simpler, still suffer from information loss due to snapshot discretization and may struggle with irregular event intervals, similar to the earlier static implementations. Additionally, many of these models rely on task-specific encoders that are not easily generalizable to new datasets or tasks.

Building on these ideas, our work proposes an alternative pathway that draws on the well-established classical static GNNs to interpret a transformed representation of the original dynamic graphs. The idea is to preserve temporal ordering and event dependencies, effectively capturing the advantages of continuous-time models, while discretizing the interactions in a controlled manner to reduce complexity and computational cost, as in discrete-time models. By unifying these aspects, we aim to retain crucial temporal information for tasks such as Dynamic Link Prediction, while benefiting from the simplicity, flexibility, and generalizability of static GNN architectures. To achieve this, we adopt a Supra-Adjacency representation, an approach introduced by Oettershagen, Kriege, Morris, and Mutzel [34] and subsequently applied in unsupervised settings by Sato et al. [42] and by Piaggesi and Panisson [37]. By mapping the input graph into a static structure through a specific transformation process, the initial temporal dependencies are directly incorporated into the graph's topological core. The resulting static model thus inherently encodes the temporal ordering within its connections and their directionalities.
However, this approach introduces certain trade-offs: while simpler model architectures become feasible, this comes at the cost of a significantly increased graph size and an additional layer of complexity in transforming the original temporal graph. Moreover, our focus on these studies shifts toward a supervised setting, which requires careful management of temporal dependencies to prevent any risk of data leakage.

In view of the aforementioned analysis, this thesis focuses on the implementation and evaluation of this method, particularly guided by the following research questions:

RQ1: Temporal Consistency. How can we construct a mapping from

continuous-time temporal graphs to static representations that strictly preserves the chronological order of events, thereby preventing data leakage and ensuring the validity of the predictive model?

RQ2: Complexity Trade-offs. What are the implications of this approach regarding computational resources? Specifically, does the reduction in architectural complexity offered by static GNNs justify the increased computational cost associated with processing the larger, mapped input graphs?

RQ3: Methodological Efficacy. Can a Supra-Adjacency representation effectively encode temporal dependencies into a static topological structure to solve supervised tasks (in our particular case a Dynamic Link Prediction), with performance comparable to native dynamic models?

Having established the main objectives of this work, we now outline the contributions developed in this thesis:

- We have adapted the Supra-Adjacency Framework to address a Dynamic Link Prediction task in a supervised setting.
  To this end, we constructed a robust pipeline that ensures the complete translation of temporal edges into a static format, generates coherent negative edges, and adapts the original task to the transformed data;

- To prevent data leakage in the supervised context, we introduce two graph representations that, by managing the allocation of edges inherently shared across different temporal splits, ensure a correct separation of variables;

- In the experimental setting, we introduce an additional "warm-up" module that operates on all nodes not involved in the learning phase. This module is carefully designed to align with the conceptual framework we developed, in which self-edges represent the temporal evolution of individual nodes, while event edges capture the original interactions;

- After implementing all feasible improvements, we conducted a systematic ablation study on the Wikipedia dataset, demonstrating that highly sparse static representations pose significant challenges to standard GNN encoders. Based on these results, we establish a more competitive "decoder-only" baseline, providing a foundation for future work, such as static-dynamic hybrid architectures.

Hereafter, we briefly illustrate the structure of this work:

- **Chapter 2 - Background:** Introduces static and temporal graphs, including definitions, historical context and applications. It finally introduces the original Supra-Adjacency matrix by Oettershagen et al. [34]

and its modification by Sato et al. [42], which form the study's starting point;

- **Chapter 3 - Methods:** Presents the core of the study. It details the transformation to the static graph, the handling of positive and negative edges, the adaptation of dynamic link prediction to the supervised context, and the methodologies implemented to prevent data leakage;
- **Chapter 4 – Experimental Setting:** Applies the methodologies to the Wikipedia dataset from TGB, testing four approaches. After limited results, a node embeddings ablation study is performed, leading to the final decoder-only approach, intended as a baseline for future work;
- **Chapter 5 – Results:** Presents results for each experimental variant, analyzing outcomes and justifying applied refinements;
- **Chapter 6 - Conclusion:** Summarizes the work and addresses the research questions.

In summary, in this thesis we explore the feasibility of applying classical GNNs to dynamic graphs by transforming the original temporal graph into a static representation. Our goal is to investigate whether static models, when combined with strategies for mapping dynamic to static graphs, can effectively capture temporal patterns without relying on fully dynamic GNN frameworks. Along the way, we discuss the main challenges inherent to this approach and identify potential strategies for developing more effective models.

Chapter 2

Background

In this chapter, we introduce the concept of graphs and discuss their relevance. We then define the essential building blocks of graph theory required for the subsequent discussion, beginning with the basic definition of a graph and its fundamental components, followed by an introduction to heterogeneous and bipartite graphs. Thus, we present the main tasks performed on graphs, focusing on node classification and link prediction.

This initial motivational and definition section is followed by an overview of the key architectures developed to operate on graphs: Graph Neural Networks (GNNs). After a brief historical review, we focus in particular on the Message-Passing Neural Network (MPNN) framework and one of its notable representative, GraphSAGE.

Finally, the chapter concludes with a description of the category of graphs and corresponding architectures central to this study: temporal graphs. We provide their definitions, a concise historical context and introduce the Supra-Adjacency representation, which is specifically applied in our work.

Graph Representations: Relevance and Applications

Graphs represent effective structures for modeling complex systems. Whenever multiple elements interact, a graph provides a clear and intuitive model of these relationships. While simple systems may be captured with direct or minimal representations, more complex systems often require network-based representations to reveal their essential structure, as seen in domains such as network science, social networks, and biology [1, 18, 29]. Graph structures support a variety of tasks: at the node level, classification or anomaly detection can be performed, while at the edge level, link prediction or edge

labeling can be addressed [25, 14, 32]. Consequently, graphs provide a unifying framework in which many real-world and theoretical phenomena share intrinsic structural characteristics, exhibiting common mathematical [7] or network-related [24, 2] properties.

Recognizing their relevance, research has expanded both conceptually and in applied contexts, as well as in the architectures developed to operate on graphs [46], ensuring that a variety of data and research questions can be effectively modeled and analyzed.

Graph Theory Fundamentals

After discussing the relevance of graphs, we now proceed to introduce some formal definitions.

A **Graph** is a mathematical structure used to model relationships between objects. Formally, a graph G is defined as a pair [7, 53]:

$$G = (V, E)$$

where:

- V is the set of **vertices** (or **nodes**), representing entities or objects;
- E is the set of **edges**, representing relationships or connections between nodes.

Edges can be described as:

- **Directed** or **undirected**, depending on whether they impose an orientation on the connection between nodes;
- **Weighted** or **unweighted**, depending on the presence of an associated numerical value.

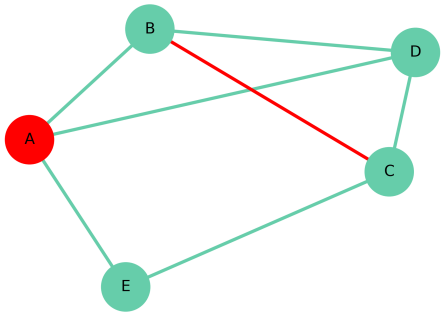


Figure 2.1: Graph representation of an undirected and unweighted graph. Node (A) and edge (B, C) are highlighted in red.

Two other definitions which are relevant are:

- **Heterogeneous Graphs:** A graph $G = (V, E, \mathcal{T}_V, \mathcal{T}_E)$ containing multiple types of nodes and/or edges, where $\mathcal{T}_V$ is the set of node types and $\mathcal{T}_E$ is the set of edge types. This formulation allows modeling complex systems with diverse entities and relationships.
- **Bipartite Graphs:** Graphs in which the nodes can be divided into two disjoint sets such that each edge connects a node from one set to a node in the other. Formally,

$$G = (V, E) \text{ is bipartite if } \begin{cases} V = U \cup W, \\ U \cap W = \emptyset, \\ E \subseteq U \times W \end{cases}$$

  It is worth mentioning that a bipartite graph is a special case of a heterogeneous graph with exactly two node types and a single edge type.

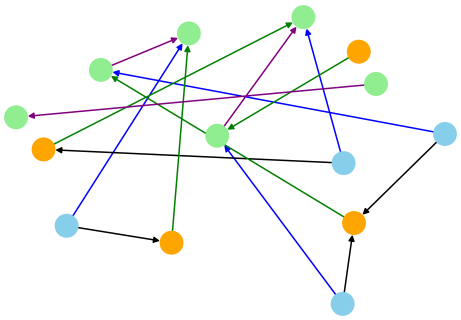


Figure 2.2: Heterogeneous graph example.

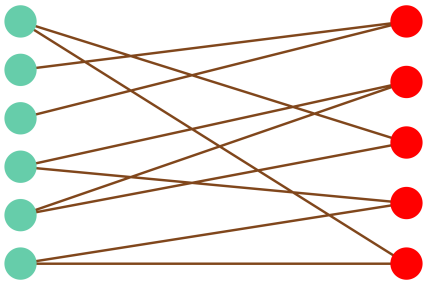


Figure 2.3: Bipartite graph example.

Graph-Based Learning Tasks

Two fundamental learning tasks commonly addressed in graph-based machine learning are [25, 15, 56]:

- **Node Classification:** The objective is to infer node labels from a subset of vertices with known values. The model employs both the graph structure and available node or edge features to infer the labels of the remaining nodes;

- **Graph Classification:** The task consists in assigning a single label to an entire graph. The model aggregates information from all nodes and edges, capturing their overall relationships to generate a comprehensive graph-level representation, which is then used to predict the graph's label.

- **Link Prediction:** Link prediction aims to estimate whether a pair of nodes will form an interaction, with models typically exploiting connectivity patterns and structural proximity to identify missing links or forecast future connections. As this is the only task applied in our work (see Section 3.2), we provide a brief overview of the methodology, focusing on the most common one for a supervised context: link prediction with negative sampling [3].

  In this approach, effective training requires both positive examples (existing edges) and negative examples (non-existent edges). Negative sampling is used to generate these non-existent edges, providing the model with contrasting cases to learn from. This procedure is particularly relevant in sparse graphs, where the number of potential negative edges substantially exceeds the positives, and it also helps accelerate training by ensuring a balanced dataset. Consequently, selecting an appropriate set of negative edges is essential for achieving reliable and robust model performance (see Paragraph 3.2 and Appendix B).

Graph Neural Network

Having established fundamental concepts, including the theoretical definition of a graph, some relevant variants, and the principal tasks that can be performed on it, we now turn to the principal models designed to operate on such structures: Graph Neural Networks (GNNs).

The concept of Graph Neural Networks (GNNs) was first formalized by Scarselli et al.[43], providing a foundational framework for learning over graph-structured data. This was followed by the introduction of Graph Convolutional Networks (GCNs) [25], which extended convolutional operations from Euclidean domains to graphs. GCNs perform localized, layer-wise propagation by aggregating feature information from a node's neighbors, allowing each node to update its representation based on its own features as well as the graph's structural context. This approach enabled efficient semi-supervised learning on graphs and paved the way for a broad range of subsequent graph neural architectures.

The general **Message-Passing Neural Network (MPNN)** framework [12] further unified these approaches, defining a flexible paradigm in which each node $v \in V$ maintains a hidden state h_v^t at iteration t. This representation is updated at each step by aggregating information from its neighbors $\mathcal{N}(v)$ through message functions. Formally, the framework can be divided into two stages:

- **Message-Passing Phase:** For a node v, the message aggregation and state update are defined as

$$m_v^{t+1} = \sum_{u \in \mathcal{N}(v)} M_t(h_v^t, h_u^t, e_{uv}), \tag{2.1}$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}), \tag{2.2}$$

  where M_t is a message function that computes information from a neighbor node u and the interaction edge e_{uv}, while U_t is an update function that combines the node's previous state with the aggregated message.

- **Readout Phase:** After T iterations, a readout function R is applied to obtain a graph-level representation:

$$\hat{y} = R(\{h_v^T \mid v \in V\}). \tag{2.3}$$

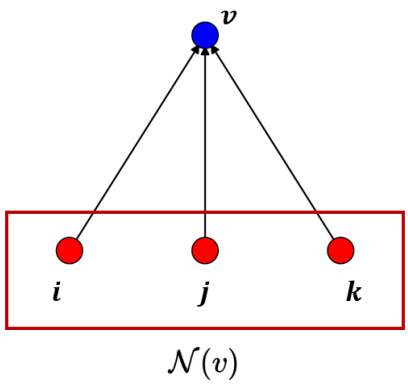
This iterative aggregation allows each node to capture increasingly global structural information while retaining the versatility to encode features from both nodes and edges.

Within this framework, **GraphSAGE** [14] is a noteworthy example that we will use, which introduced an inductive approach that enables efficient representation learning on previously unseen nodes by sampling and aggregating features from local neighborhoods.
For a node v, its representation at layer $k+1$ is computed as

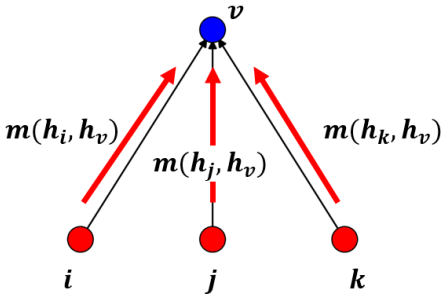
$$h_v^{k+1} = \quad \sigma\Big(W^k \cdot \text{AGGREGATE}\big(\{h_v^k\} \cup \{h_u^k \mid u \in \mathcal{N}(v)\}\big)\Big), \tag{2.4}$$

where

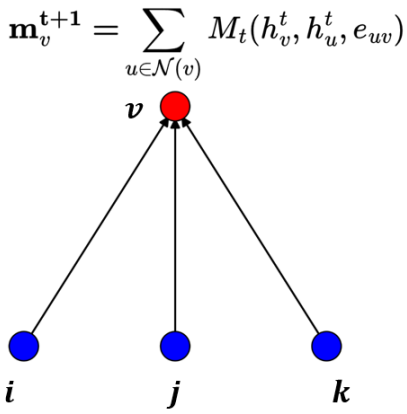
- **AGGREGATE** is a permutation-invariant function (e.g. sum, mean, max, or LSTM-based aggregation):
- $\mathbf{W^k}$ is a learnable weight matrix at layer k;



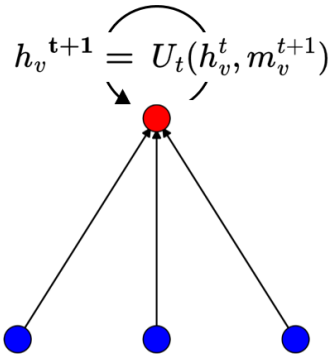
(a) Node v and its neighbours $\mathcal{N}(v)$



(b) Message creation step



(c) Message aggregation step



(d) Representation update step

Figure 2.4: MPNN - Message-Passing Phase steps.

– σ is a non-linear activation;

– $\mathcal{N}(v)$ denotes the neighbors of v.

This inductive mechanism allows GraphSAGE to generalize to nodes not seen during training, improving scalability and applicability to large graphs.

Subsequent developments, including Graph Attention Networks (GATs) [50], further enhanced the expressiveness of GNNs by introducing attention mechanisms. These allow nodes to value the importance of their neighbors differently during aggregation, providing the means for the model to focus on

the most relevant parts of the graph structure. This improves learning capacity on heterogeneous or noisy networks and represents significant theoretical and practical advancement in the field.

Dynamic graphs

In recent years there has been growing interest in a precise typology of graphs known as **Dynamic (or Temporal) graphs**. Their emergence has become necessary because even events that could previously be represented with simple vectors or matrices acquire substantially greater complexity when time is taken into account. This increased complexity makes traditional modeling assumptions outdated and demands new approaches to address these challenges [35, 41]. Classical representations are unable to adequately capture the evolution of both parameters and structure, an essential capability for studying real-world phenomena, ranging from biology and disease propagation [18] to social media interactions [8] and fraud detection [40].

Since our study operates within this well-defined context, it is important to review previous works in the field in order to understand the crucial aspects that must be considered throughout our discussion. First, a temporal graph is formally defined, followed by an introduction of the Dynamic Link Prediction task as a necessary foundation for the discussion; subsequently, the history of its representations and the methods developed to analyze them are presented.

Definition (Dynamic Graphs) A dynamic (or temporal) graph differs from a classical (static) graph (Eq. 2) by the presence of an additional temporal component [39, 16]. We define it as

$$G = (V, E, T), \tag{2.5}$$

where V is the set of vertices, E is the set of edges, and T is the set of timestamps where the edges exist, the key notion underlying temporal graphs (Fig. 2.5).
In particular we define each edge as

$$e = (u, v, t) \in E, \tag{2.6}$$

i.e. e is a temporal edge from vertex u to vertex v that occurs at time t. Finally, we define $v \in V$ to be active if there exists at least one temporal edge that starts or ends at v.

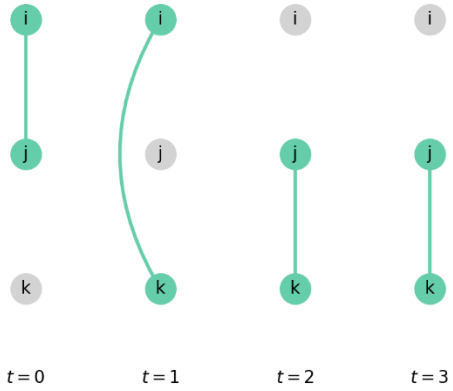


Figure 2.5: Example of a Temporal graph.

Definition (Dynamic Link Prediction) Given a temporal graph $G = (V, E, T)$, the Dynamic Link Prediction task consists in predicting whether a link (u, v) will appear at a future time $t' \in T$. Formally, the goal is to learn a function

$$f : (u, v, t_{\text{hist}}) \mapsto \hat{y}_{uv}(t')$$

where $\hat{y}_{uv}(t') \in [0, 1]$ represents the predicted probability that an edge between nodes u and v will exist at time t', conditioned on the observed history of the temporal graph up to time $t_{\text{hist}} < t'$.

Dynamic Link Prediction generalizes the classical link prediction (2) problem by incorporating the temporal evolution of edges.

Moving forward, a temporal graph can be represented in various ways, and the models applied to these representations have their own strengths and drawbacks [22], which are important to understand.
To continue the discussion on the evolution of temporal graphs and graph neural networks, we draw inspiration from the work of Longa et al. [28].

An early and straightforward approach was to represent temporal graphs using static structures. However, this approach fails to capture the rich temporal information inherent in the data: the graph is represented only by source–destination pairs, completely ignoring the temporal component. While classical GNN architectures can be applied to such static representations, this simplification inevitably results in a significant loss of information and prevents the model from accounting for the dynamics introduced by the temporal dimension. Shortly after this initial treatment, two other representations emerged in parallel: Discrete-Time Dynamic Graphs (DTDGs),

to which snapshot-based models are applied, and Continuous-Time Dynamic Graphs (CTDGs), which are instead processed with event-based models.

To be more precise, DTDGs consist of a series of static graphs, each collecting all the edges that occur within a specific temporal slot[1]. The most widely recognized models operating on this representation are EvolveGCN [35] and DySAT [41], which extend classical GNN architectures to capture the evolving nature of the graph across snapshots. Despite introducing temporal knowledge into the modeling process, the snapshot-based approach still fails to fully exploit the temporal dynamics due to the inherent fragmentation in its definition.

On the other hand, CTDGs are represented as a list of timestamped edges. A notable early attempt to address dynamic graph learning in this event-based setting was the Temporal Graph Network (TGN) [39], which introduces a dedicated memory module to capture each node's historical state and injects it, along with the structural information collected, into a standard GNN architecture. Several alternative approaches have since been proposed: DyRep [49], DygFormer [54], and NAT [31], which focus on neighbor interactions; CAW [51] and Base3 [26], which rely on intrinsic graph motifs and EdgeBank [38], which adopts a direct edge-based memory approach, just to name few of them. Each of these models addresses the full dynamics of graphs in different ways, highlighting the diversity of approaches in temporal graph analysis and the various factors that must be considered: it is evident that certain methods perform well under some criteria but fail under others [38, 22].

It is important to note that methods for DTDGs and CTDGs treat temporal and structural information separately. Structural analysis typically examines features such as node neighborhoods and network motifs (e.g. triads, edge patterns, centrality, co-occurrence frequency [34, 31]), whereas temporal analysis captures aspects like event decay over time or node activity frequency [30, 11]. Eventually, these two types of information are integrated and processed by a variety of architectures, ranging from Transformers [52, 30] to simple MLPs [20, 22] (or in some cases, used directly without further modeling) to accomplish the target task.

To conclude, the final category of representations we discuss and adopt in our work fuses static and temporal information into a single topological structure. The core idea is to construct a static representation of the graph, which allows the use of classical GNNs, as they are well-established and computationally efficient, while still preserving the full temporal knowledge

[1]An alternative definition exists but is less frequently used: in this version, each snapshot collects all edges up to a given timestamp (cumulative snapshots).

embedded within the graph.

2.1 Supra-adjacency Matrix

There exist multiple definitions of a supra-adjacency matrix [34, 42]. In this work, we propose a formulation that integrates all essential aspects and modifications considered necessary for our analysis. For clarity we present a detailed formulation below, starting from the elementary variables.

"[...] Temporal networks can be seen as a special multilayer network in which every layer coincides with a timestamp t"[10].
Therefore, defined a temporal graph $G = (V, E, T)$, we first introduce the following definitions, adopting a notation similar to that used in [34]:

- T_ω, the set of times where edges are incident to ω (i.e. the timestamps when we define ω as active);
- $\tau_\omega(i)$, the index in T_ω where we can find the timestamp i:

$$T_\omega = \{t_{\omega,a} \in T \mid \exists(\omega, j, t_{\omega,a}), \quad \omega, j \in V, t_{\omega,a} \in T, a \in \{0, ..., K\}\}, \tag{2.7}$$

$$T_\omega[\tau_\omega(i)] = i, \tag{2.8}$$

where $t_{\omega,a}$ in the a_{th} timestamp of ω. It is immediate to see that $T_\omega \subseteq T$.
Note that, while it is true $t_{\omega,a} = t_{j,b}$ when there exists $(\omega, j, t_{\omega,a}) = (\omega, j, t_{j,b})$, the same cannot be said for $t_{\omega,a}$ and $t_{j,a}$ (and analogously for index b) because a priori we have no information about the temporal ordering of the next or previous interactions in which either ω or j participate, and therefore their respective subsequent timestamps t_s.

Considering these definitions we can describe the core idea of the Supra Adjacency matrix which maps temporal edges $E = \{(u, v, t) \in E \mid \{u, v\} \in V, t \in T\}$ into a set of edges, each having a precise definition inside the thought process. For the sake of completeness, and since the second definition is derived from a simplified version of the first, we will present both. However, in the subsequent discussion, we will use only the more concise second one.

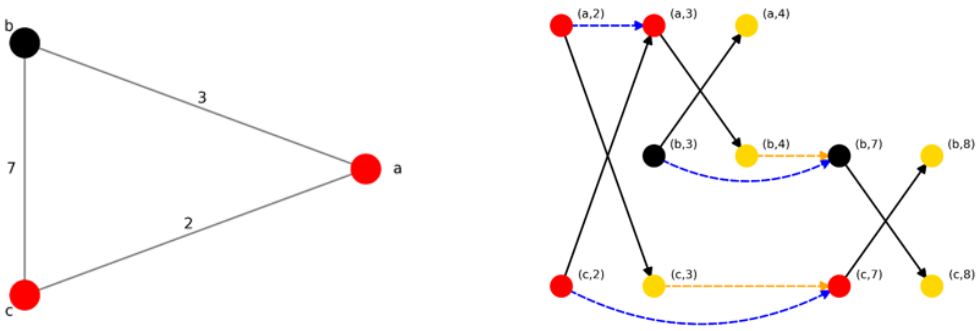
2.1.1 Static Expansion: Additional nodes model

In this first case we adopt the definition proposed by Oettershagen, Kriege, Morris, and Mutzel [34]: All the original nodes of the temporal graph $G = (V, E, T)$ will be mapped into tuples of the kind $(node, timestamp) \in U$. Likewise, the temporal edges $e = \{(u, v, t) \in E,\ u, v \in V, t \in T\}$ will be mapped into three kinds of edges that are defined in the following way:

$$
\begin{cases}
E_N = \big\{\big((u,t),(v,t{+}1)\big),\ \big((v,t),(u,t{+}1)\big) \mid (\{u,v\},t) \in E\big\} \\
E_{W_1} = \big\{\big((w,i{+}1),(w,j)\big) \mid (w,i{+}1),(w,j) \in U,\ i,j \in T(w), \\
\qquad\qquad \tau_w(i)+1 = \tau_w(j),\ i+1 < j\big\} \\
E_{W_2} = \big\{\big((w,i),(w,j)\big) \mid (w,i),(w,j) \in U,\ i,j \in T(w), \\
\qquad\qquad \tau_w(i)+1 = \tau_w(j),\ i < j\big\}
\end{cases} \tag{2.9}
$$

where E_N represent the edges bringing the original temporal edge information (u, v, t), E_{W_1} are the edges which connect additional nodes necessary to build the E_N edges to the subsequent effective nodes (Fig. 2.6); E_{W_2} connect subsequent effective nodes. In summary, E_N carries the original temporal edge information (i.e. the interaction ground truth e), while E_{W_1} and E_{W_2} preserve the temporal ordering by linking the $(\text{node}, \text{timestamp})$ tuples that correspond to one point of the temporal evolution of the same node $\in V$ in the temporal graph.

This definition was originally developed for a Dissemination process [34], specifically for undirected graphs (i.e. $(u, v, t) = (v, u, t)$), which in certain cases leads to the creation of additional edges. A further remark is that the model currently operates without node or edge features. Nonetheless, this limitation is not critical for our initial objectives and can be addressed with minor modifications, which we will detail in the following sections.



(a) Temporal Graph (b) Static expansion

Figure 2.6: A temporal graph and its static expansion representation. In the right figure, black edges belong to E_N, yellow edges to E_{W_1}, and blue edges to E_{W_2}. Yellow nodes indicate the additional vertices introduced in the expansion.

2.1.2 Modified Static Expansion: No Fictional nodes model

Henceforth, the approach described corresponds to that proposed by Sato et al. [42], which defines that, given a temporal graph $G = (V, E, T)$ with edges of the form (i, j, t), its static version can be represented by replacing each node with a tuple $(\text{node}, \text{timestamp})$ and each temporal edge with two types of edges (Fig. 2.7):

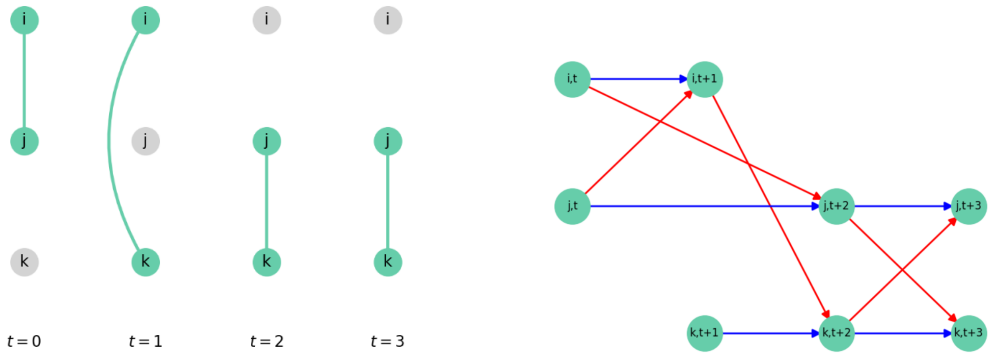
$$\begin{cases} E_{\text{event}} = \Big\{ \big((i, t_{i,a}), (j, t_{j,b+1})\big),\ \big((j, t_{j,b}), (i, t_{i,a+1})\big) \ \Big|\ (i, j, t) \in E, \\ \qquad\qquad i, j \in V,\ t_{i,a}, t_{i,a+1} \in T_i,\ t_{j,b}, t_{j,b+1} \in T_j \Big\}; \\ E_{\text{self}} = \Big\{ \big((i, t_{i,a}), (i, t_{i,a+1})\big) \ \Big|\ i \in V,\ t_{i,a}, t_{i,a+1} \in T_i \Big\}. \end{cases} \tag{2.10}$$

In this formulation, only the effective nodes are used to construct the edge sets. As a consequence, the introduction of an auxiliary set such as E_{W_1}, together with its additional intermediate nodes, becomes unnecessary. Instead, the set E_{event} can be interpreted as a modified counterpart of the previously defined E_N, while E_{self} plays a role analogous to E_{W_2} ensuring temporal consistency along each node's evolution.

Their names are in fact not coincidental: E_{event} refers to the edges which represent the events encoded in the original temporal graph, whereas E_{self} denotes the self-referential connections of a node (i.e. its temporal evolution), which ultimately form long chains that we will later exploit in the implementations discussed in Section 4.3.

Note that, as in the previous subsection 2.1.1, we consider undirected edges and which do not incorporate node or edge features.

Nevertheless, both constructions remain only partially correct. In the following chapter, we show what are their specific limitations and propose an approach to mitigate them.



(a) Temporal Graph

(b) Modified static expansion

Figure 2.7: A temporal graph and its modified static expansion representation. In the right figure, red edges belong to E_{event} while blue ones belong to E_{self}.

Chapter 3

Methods

In this chapter, we present all the logical steps required to obtain an actual static representation. We begin by refining the Modified Static Expansion described in paragraph 2.1.2, introducing one or more additional layers necessary to achieve a complete mapping of temporal edges. After the mapping of positive and negative edges, we proceed defining the two types of static graphs used in our framework. Finally, we conclude with the ID mapping step, which is required to make the resulting structure compatible with the architectures implemented in the following section.

3.1 Observation Layer Representation

As anticipated, the two approaches previously introduced in Sections 2.1.1 and 2.1.2 exhibit two major shortcomings. Although the work of Oettershagen, Kriege, Morris, and Mutzel [34] shows the Static Expansion they propose fully preserves the information encoded in the original temporal graph, the resulting representation may be unnecessarily cumbersome.

In particular, it requires the introduction of additional nodes and an auxiliary edge set E_{W_1}, whose sole purpose is to track their relationships. Conversely, the lighter version of the Modified Static Expansion as currently defined does not guarantee the complete transfer of temporal information throughout the model.

To clarify this second issue and the resulting proposed solution, which was developed independently but can upon further consideration, be seen to have some conceptual relation to Section 2.1.1, we introduce a simple temporal graph along with its corresponding Modified Static Expansion (see Section 2.1.2). Consider the temporal graph $G = (V, E, T)$, assumed undirected for simplicity, with:

$$V = \{i, j, k\};\ T = \{0, 1, 2, 3\};\ E = \{(i, j, 0), (i, k, 1), (j, k, 2), (j, k, 3)\}.$$

Its representation and corresponding modified static expansion are presented in Fig. 3.1 and Fig. 3.2.

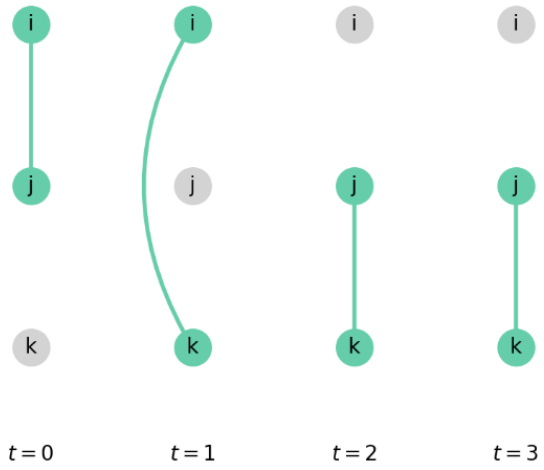


Figure 3.1: Example of temporal graph.

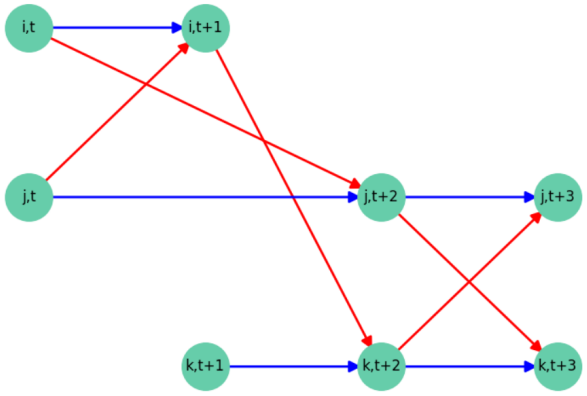


Figure 3.2: Modified static expansion of Fig. 3.1.

If we follow the definition in Eq. 2.10, each temporal edge is expected to generate a quartet of static edges: two event edges and two self-edges. However, as illustrated in Fig. 3.2, the event edge that should originate from $(k, t+1)$ is missing because no corresponding terminal node exists at $(i, t+2)$.

As a result, the modified static expansion of the temporal edge $(i, k, 1)$ remains incomplete.
It should be noted that this issue, the missing edges resulting from the original formulation, cannot happen in the richer definition reported in paragraph 2.1.1.

The difficulty arises because the model as it is defined preserves all the original information but is unable to express it fully (i.e. not all mapped edges can be constructed). To overcome this limitation, we introduce an additional layer of nodes in the temporal graph representation which are considered active one timestamp after the last available $t \in T$ of the original model (Fig. 3.3).

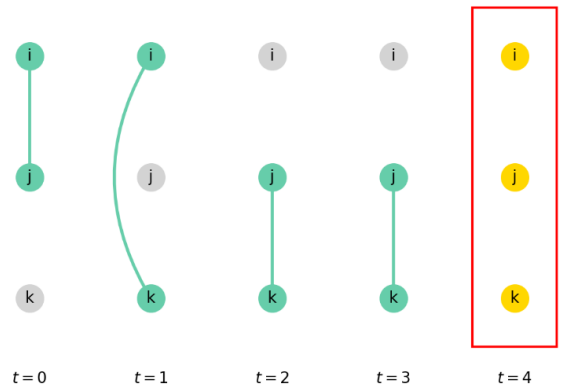
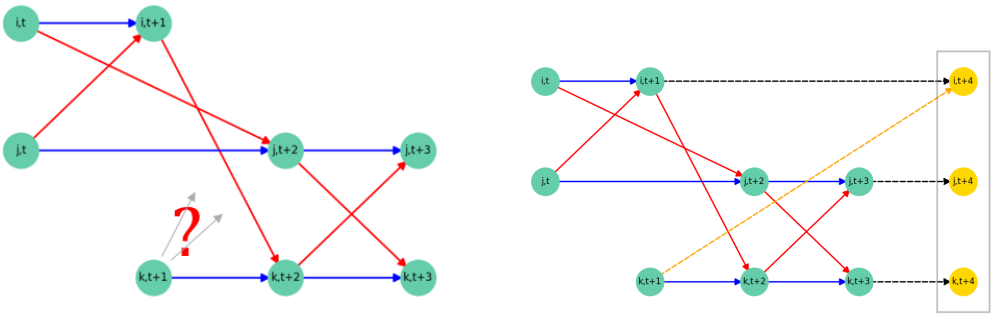


Figure 3.3: Temporal graph with additional Observation Layer.

Formally, we treat these nodes as isolated (i.e. they do not participate in any temporal edge, whether belonging to the original edge set E or otherwise). In their corresponding mapping into the modified static expansion (Section 2.1.2), they act as structural placeholders that facilitate the transformation of each temporal edge into its triplet or quartet based representation. Additionally, they enable the inclusion of a final layer of self-edges, providing a concluding observation of each node at the end of the event sequence.

This mechanism ensures the information and its transmission are fully defined and preserved, without altering the relationships established in Eq. 2.10. In fact, the only changes are in the number of nodes and edges in G, which now include all the properly mapped elements along with the final layer of self-edges.

Given the discussion above, we refer to this final layer of nodes as the Observation Layer, since it represents the state of each node observable after all events have occurred. This conceptual idea is necessary for the two



(a) Modified static expansion representation, one mapping edge cannot be created.

(b) Observation Layers representation in the Modified Static Expansion representation.

Figure 3.4: Example of the Observation Layer representation.

methodologies we will briefly introduce, but it is particularly crucial for the development of the second (see Section 3.3.2).

It is also worth noting this concept is not entirely dissimilar to the additional nodes introduced in Section 2.1.1. However, in this case we carefully select both the number and configuration of nodes required, which in turn reduces memory usage, and treat them as an effective component of our temporal graph, assigning them an actual role (i.e. the Observation phase) and allowing us to merge the previous edge set E_{W_1} into the single edge set $E_{self} \sim E_{W_2}$.

Finally, alternative design choices are possible. We adopt this formulation because it aligns naturally with a Temporal Splitting strategy, which is commonly used when reporting supervised temporal graphs and does not introduce significant downstream effects. In contrast, Random Splitting may require a different treatment, potentially suggesting new directions for future research.

3.2 Dataset Transformation

Positive edges

With the refined temporal graph by incorporating a final layer of Observation nodes defined, we can now proceed to convert the original set of temporal edges into its modified static expansion.
Unlike the original given definitions (Eq. 2.10), we will treat more in detail the case of a bipartite and directed graph: For this reason each temporal

edge will be mapped not into a quartet of edges (i.e. two event and two self-edges) but into a triplet one (i.e. one event and two self-edges) since we do not have to deal with the bidirectionality of the interactions.

Given the Temporal graph $G = (V, E, T)$, its corresponding mapped version can be defined as $G_{map} = (V_{map}, E_{map})$, where $V_{map} = V_{source,map} \cup V_{destination,map}, V_{source,map} \cap V_{destination,map} = \emptyset$ made of $V_{i,map} = \{(v,t) \mid v \in V_i, t \in T_v, i = \{source, destination\}\}$ and the original $e \in E$ are mapped in the following three types of independent edges sets $e_{event} \in E_{event}, e_{self,source} \in E_{self,source}$ and $e_{self,destination} \in E_{self,destination}$, defined in the following way:

$$
\begin{aligned}
E_{event} &= \Big\{((i,t),(j,t+\delta t_j)) \;\Big|\; (i,j,t) \in E,\ i,j \in V, \\
&\qquad t \in T_i \cap T_j,\ t+\delta t_j \in T_j,\ T_j[\tau_j(t)+1] = t+\delta t_j\Big\}, \\
E_{self,source} &= \Big\{((i,t),(i,t+\delta t_i)) \;\Big|\; i \in V, \\
&\qquad t, t+\delta t_i \in T_i,\ T_i[\tau_i(t)+1] = t+\delta t_i\Big\}, \\
E_{self,destination} &= \Big\{((j,t),(j,t+\delta t_j)) \;\Big|\; j \in V, \\
&\qquad t, t+\delta t_j \in T_j,\ T_j[\tau_j(t)+1] = t+\delta t_j\Big\}.
\end{aligned}
\tag{3.1}
$$

where we used $\delta t_{i,j}$ to describe the increments in time which are not necessarily equal to 1 like in subsection 2.1.1, but are generated according to the time difference between the timestamps where a certain node is active. As we follow the given definition of 3.1, to not miss any kind of edge, we generate and add a timestamp layer of active nodes at $t_{obs\_layer} = max(t) + 1, t \in T$ (i.e. $(node, t_{obs\_layer})$ in our case) and we use it to complete the translation of the various temporal edges. In this way we have created properly the full static graph we can proceed with.

It is important to emphasize the edges e_{event} constitute the true translation of the original temporal interactions (i,j,t). In contrast, the edges $e_{self,source}$ and $e_{self,destination}$ encode the temporal evolution of the nodes i and j, respectively, across successive timestamps.
Finally, the extended process of how we to practically execute this translation is described in appendix A.

Link Prediction

A further question to address is how the original Dynamic Link Prediction task is translated within our framework, which consists of evaluating whether a given source and destination will interact at a specific timestamp (see paragraph 2).
If at first a linear transposition would lead us to execute a link prediction for all the three (or four if the graph is undirected) kinds of edges, then we can derive, due to the nature of our transposition (Section 2.1.2) and the conceptual information of the various sets in relation to the original temporal edges, that our task can be reduced to a Link Prediction only on the event edges E_{event} (conceptually equivalent to the E_N set of Section 2.1.1). Indeed upon closer examination E_{event} is the true provider of the full original information. In contrast, $E_{self,source}$ and $E_{self,destination}$ (conceptually equivalent to the E_{W_1} and E_{W_2} sets from Section 2.1.1) carry only a partial information of the actual interactions and exist primarily for structural-temporal ordering. We will revisit and extend the role played by these latter edges later in our work.

Negative edges

Since we are dealing with a (Dynamic) Link Prediction task (section 2) we have to handle also the existence of negative edges. In general, datasets provide a negative sampler and auxiliary functions to cover this need, often including a ready-to-use set of negative edges for the specific task [4].
The literature presents various definitions for a negative edge and specifically Poursafaei et al. [38], point out how this choice strongly influences both the evaluation results and the extent to which an architecture relies on past history or the inductive capabilities provided by the sequence of temporal edges. Conscious of this previous discussion, we choose to adopt for our work the definition of a random negative edge: given a positive temporal edge (src, dst, t), the corresponding negative edge is defined as (src, dst_{neg}, t), where dst_{neg} is randomly selected and therefore, realistically non-existent at time t[1].

$$dst^i_{neg} = \{j \mid (i, j, t) \notin E \text{ and } (k, j, t') \in E, i, k, j \in V, t \in T, t' \in T_j\}. \quad (3.2)$$

where T_j is a subset of T where j is active.

[1]The variant we choose to discard is the historical one, which selects past existent source-destination pairs, that however did not occur at the timestamp under consideration. It is also relevant to note in some cases a random method could end up choosing a historical variant, although this is very uncommon.

It is worth noting that $t \in T_j$ is also possible if there exists at least one triplet (z, j, t) with $z \neq i$ and $z, i \in V$.
Secondly in a supervised setting, the set of timestamps T is replaced by T_{train} during the learning phase. Therefore, T_{train} contains only the timestamps relevant to the training split, and nodes that do not appear in the training set are excluded from being potential dst_{neg}. This ensures negative training edges are sampled only from nodes whose history is available during training, making the learning process more conservative compared to the random choice mentioned previously. In this current form, this can be viewed as a "relaxed historical" variant of negatives rather than a purely random one (see Appendix B).

The contribution of our approach lies in the actual mapping of these edges into their respective static versions, which provides an additional degree of freedom unavailable to previous methods. First, it is important to recall that this translation differs from the one applied to positive edges: in the case of negative edges, it is necessary only to translate them into their E_{event} representation, without considering the E_{self} edges of either kind. This is because we are exclusively predicting the first type of positive edges, as detailed previously in Section 3.2. Given a potential negative temporal edge (src, dst_{neg}, t) corresponding to the positive edge (src, dst, t), we have an additional degree of freedom. To introduce this new variable, we recall that a negative edge is mapped to the following (event) edge:

$$[(src, t), (dst_{neg}, t_{neg})], \tag{3.3}$$

where t_{neg} can be chosen with some freedom, constrained only by the requirement that it belongs to the set of available timestamps of the selected dst_{neg}. This restriction is introduced to avoid memorization catastrophes and to prevent the model from selecting newly created nodes, which are far more numerous than the active ones used to form the positive edges. In fact, since these (potential new) nodes are never involved in the corresponding positive translations, they would be trivial to detect, being completely isolated from the original static graph topology, and therefore not meaningful.
Accordingly with this, three options can be considered within this definition space for selecting t_{neg} and the corresponding variants of negative edges:

1. **Hard Negatives**, where t_{neg} is chosen by applying the original rules of eq. 3.1 to dst_{neg} as if it were positive;

2. **Weak Negatives**, where t_{neg} is chosen at random[1], with the only constraint that it is greater than the original t ($t_{neg} > t$);

3. **Very Weak Negatives**, where t_{neg} is chosen completely at random[1] among all available timestamps.

As their names suggest, their complexity and equivalence with the corresponding mapped positive event edge decrease along the hierarchy: Hard negatives could be actual positives if they were simply added to the original list of temporal edges; Weak negatives still preserve the temporal ordering between the source and destination while Very Weak Negatives are more structural in nature, as they are defined purely by the "negative definition" rather than by Eq. 3.1: they represent edges that can be constructed but are not positive, following no predefined rules except that they connect pre-existing node tuples.
Additionally, it is important to ensure that T_i, the set of timestamps for a given node i, is defined to contain all timestamps required to construct at least all necessary hard negative edges (i.e. at least one timestamp per node with $t > t_{max,i}$). This requirement does not apply to Weak and Very Weak negatives, as the timestamps included for Hard negatives already provide sufficient values for these cases.

Moving towards the supervised context, which will be the core of our discussion, we need to define how negative edges are constructed within each split (training, validation, and test). We choose to restrict the generation of validation and test edges, often provided by the dataset itself [4], to the Hard negative definition, while allowing the full range of the three negative edge types and their proportions to vary for the training set.
This strategy is adopted to maintain the highest degree of difficulty during the evaluation phase, as the negative edges closely resemble the positive ones, while the variability among the variants in the training phase, adjusted through two additional hyperparameters (see Appendix B), aims to foster inductive capabilities in our approach. Therefore in our discussion we generate negative edges for the training set with varying proportions among the three variants, whereas for the validation and test sets, we either translate the provided negative vectors when available or generate all edges as Hard negatives.

A more detailed explanation of the mapping of validation and test negative edges when given, as well as the generation of training negatives, is provided in Appendix B.

[1] Chosen uniformly at random among the available timestamps.

3.3 Managing Data Leakage

3.3.1 Data Splits approach

As we will later see in Chapter 4, the datasets used in the supervised context have a built-in split method that can be easily transposed. Since we know the interval to which each original edge belongs, we can place their corresponding translated edges into that same interval.

The only variation requiring special treatment occurs at the intersection of splits: if a node (either $(source, t)$ or $(destination, t)$) appears in two or more edges belonging to original temporal edges assigned to different splits, these edges must be handled carefully to avoid data leakage.

A standard approach to prevent unintended information transfer is to move all edges containing nodes shared across splits into their destination split. Nonetheless, this method introduces significant challenges in our static translation. To underline what are the potential issues and possible solutions let us analyze self and event edges singularly.

Self-Edges case

Self-edges arrangement actually does not lead to particular problems. According to the conceptual framework of self-edges, the two ends of each edge represent the same node in the original graph, observed at different timestamps. Based on this consideration, it is reasonable to think we need to place each edge on the cross of the splits on the arrival one. Indeed, it is straightforward to observe that, under this approach, the $(node, timestamp)$ with the smaller timestamp (hereafter called also the "Past node" since it is temporally antecedent) initially gathers information from the prior events and subsequently can pass its state to its corresponding "Future" self node (i.e. the destination of the self-edge) only in the following split. To conclude, the standard approach is both applicable and necessary for self-edges.

Event Edges case

A different point of view has to be applied to manage properly the content of event edges. If we superficially allocate event edges at the cross split into the destination one we will incur into more than one difficulty: due to the (Dynamic) Link prediction (Sec. 3.2), we associate to each positive event edge a list of negative edges. As it is often the case, and as we will discuss later in Paragraph 4.1.1, we must create the training negative edges manually, whereas the validation and test edges are provided by the dataset.

Examining the latter cases, we observe that, due to our oversight, each positive event edge and its corresponding set of negative edges would need to be moved, thereby compromising the originally provided structure. Moreover, being each negative edge of event type recalling the hypothesis given in paragraph 3.2, also each Negative edge should belong to a different split according to its negative destination map.

It is evident that event edges, either positive or negative, must reside in the source node split to avoid any inconsistencies, unlike what might result from a standard approach. It is also necessary to ensure this arrangement is robust against data leakage, as failure to do so would make the model's outputs fundamentally flawed. To achieve this, we consider two distinct cases: Bipartite/Directed and Non-Bipartite/Undirected graphs. We anticipate that the trial dataset Wikipedia previously cited, falls into the first category, but we will demonstrate how this method can be applied to any type of dataset.

• **Bipartite/Directed case:** In this case, as shown in Fig. 3.5a, the "Future" node can be hit only by "Past" nodes belonging to one of the previous splits: all the edges in which this node is present are cross edges by construction rules (see Eq. 3.1). Therefore, we can be confident that the information arriving at a destination node originates from only one source split, distinct from the split in which it terminates. It is worth noting that even if an edge spans the training and test splits (i.e. there is another split in between), no validation edge could possibly connect to that destination node. Otherwise, by definition, an additional node (the white one) would exist, and all incoming edges would instead point to it. In this way the "Future" node receives only information from the "Past" self (if existent) and all the other nodes which had interacted with it in the past. Since we are considering a directed setting, each destination node is connected only to its own "future" instance through a self-edge (here corresponding to $e_{\text{self,destination}}$). In the first split, the "Future" node participates in interactions and receives information from them (in this case, it is reached by training event edges). In the subsequent split, it propagates the accumulated information to its next "future" instance. This procedure ensures that information is not revealed prematurely but is instead progressively collected and appropriately transmitted to the following step.

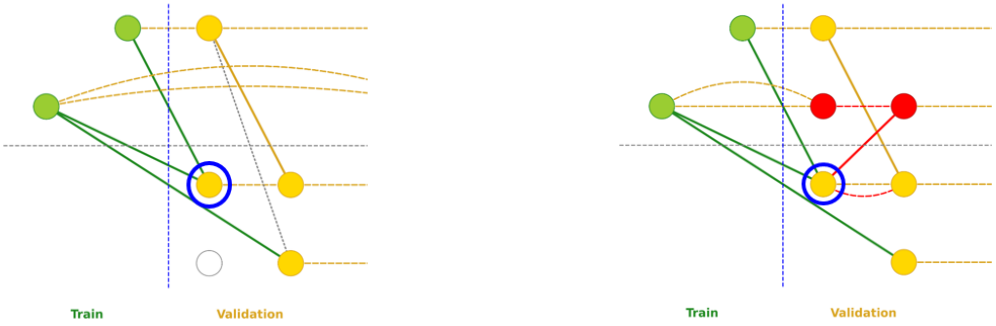
• **Undirected case:** In this case the destination node exists only if one the following two events happens: at least another node interacts with the "Future" node in a later timestamp and the node becomes

1. The "Past" node which is connected only through a self-edge if the interaction is not bidirectional (i.e. it is later reached by another source

and is not the origin of an event edge);

2. The "Past" node is connected through both a self-edge and one or more event edges otherwise (see Fig. 3.5b).

While case (1) can be directly related to the Bipartite/Directed setting (Fig. 3.5a), the second case requires a more detailed analysis. In this latter scenario, during the training phase, the "Future" node first collects information (as before, coming only from one previous split by construction) and then, becoming the "Past" node in the subsequent split (i.e. validation), it propagates its processed information to its "future" self and to the other "future" destinations of its event edges. Also in this case, the mechanism functions correctly, as the node can actively "learn" from its interactions in the "Past" split and subsequently act upon the next one without losing the notion of what occurred and how its history affects its state.



(a) Directed/Bipartite case

(b) Undirected case

Figure 3.5: Possible cases of event edges across splits. The circled node represents the main "Future"/"Past" node. Red nodes denote the auxiliary nodes forming the event edge originating from the "Past" node (Undirected, case 2). Note that cross self-edges have already been moved to their corresponding destination time split according to 3.3.1.

To summarize, the edges at the cross splits have to be moved:

1. Towards the destination split if they are self-edges;

2. Towards the source split if they are event edges.

This idea could be enforced also observing the core idea behind the tuples $(node, timestamp)$ and edge types definitions: in self-edges it is more important the destination nodes receive the information after the source has

elaborated them (i.e. the "Future" has to know about its "Past", not the opposite); in event edges instead the source nodes are the only ones which bring a piece of the original temporal edge (the original source and timestamp), while the destination loses the last notion by definition. In this case is better to keep the edge in the source and then use the destination as an information channel to the other nodes of the next split.

Finally, it is important to bring attention to another potential critical issue of this approach, which may have previously gone unnoticed: the possible scarcity of self-edges in the training and validation phases. If a node participates only once or twice in a temporal edge within a given split, it will consequently be associated with only one or two self-edges, one of which, by construction, lies at the boundary between two splits and is therefore re-assigned to the destination split according to the previous discussion. As a result, the earlier split (in particular the training split, which unlike validation does not receive cross-split self-edges from a preceding interval) may become depleted of most of its self-edges. This leads to an imbalance and a substantial reduction in the amount of information available to propagate through the mapped static graph, with potentially severe consequences.

3.3.2 Observation Layers approach

As discussed in the previous paragraphs, the original methods proposed in [34, 42] required progressive adjustments to be fully applicable to a supervised learning setting. Building on these observations, we developed an initial methodology in Section 3.3.1, which relies on a precise management of cross-edge re-allocation among temporal splits. However, since we aim for a more robust and conservative approach, one that protects against any form of data leakage and completely eliminates the self-edge depletion that the previous model may experience, we now introduce an extension of the previously defined concept of observation layers (see paragraph 3.1) to each temporal split.

Thus far we have introduced an additional observation layer temporally posterior to the whole series of actions: the static representation [34] reported in section 2.1.2 was able to save the information of the original temporal graph but it was not equally effective to express them fully without this addition (see Fig. 3.4b). As the name refers to, the observation layer is not an "active" state for the nodes (i.e. we do not have any kind of actions happening in the specific temporal moment in which we generate the layer), nonetheless this is a stage in which the global system is analyzed, allowing us to identify for each node the effects of a collective set of transformations applied to it.

The principle of the observation layer is non-interference with the sequence of temporal edges. This allows us to define as many observation layers as needed, provided that their definition lies outside the discrete time points T at which the nodes actually interact. For our purposes, we introduce two additional layers on top of the final one, resulting in a total of three observation layers in the global assessment (i.e. one for each time split).
More formally, we define the set of nodes constituting these observation layers as follows:

$$\begin{aligned} \textit{Observation Layer} = \{(node, t_{obs}) \mid \exists (node, t) \in U,\ (node, t_{obs}) \notin U \\ \text{and}\ \ t < t_{obs}\}. \end{aligned} \tag{3.4}$$

For our problem we have to restrict the range of available t_{obs} to three elements:

$$t_{obs} \in T_{obs\_lyrs} = \{t_{\max,train} + \delta t_{train},\ t_{\max,val} + \delta t_{val},\ t_{\max,test} + \delta t_{test}\} \tag{3.5}$$

where $\{t_{\max,train}, t_{\max,val}, t_{\max,test}\} \in T$ are the biggest timestamps available for each specific time split and

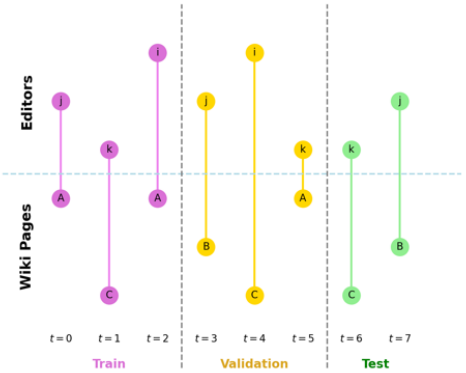
$$\begin{aligned} t_{\max,i} < t_{\max,i} + \delta t_i < t_{\min,j}, \\ t_{\max,\text{test}} < t_{\max,\text{test}} + \delta t_{\text{test}} \end{aligned} \tag{3.6}$$

where

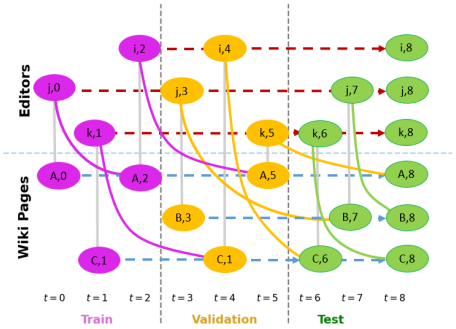
$$i \in \{\text{train}, \text{val}, \text{test}\}, \quad j \in \{\text{train}, \text{val}, \text{test}\}, \quad i < j$$

i.e. only the combinations train–val, train–test, and val–test are considered.

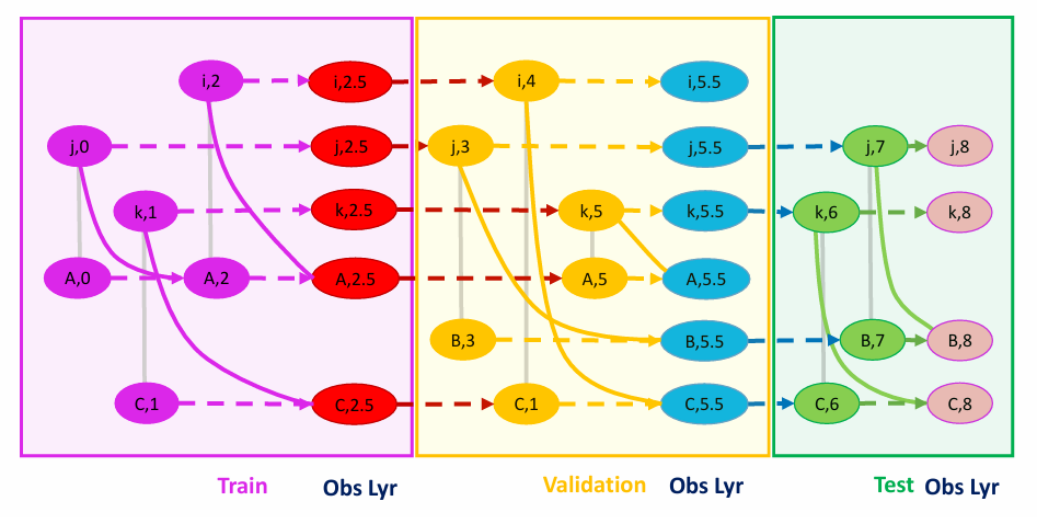
These additional nodes are added into G_{map} and must be incorporated into the original construction (Eq. 3.1): the timestamps associated with each new $(\text{node}, \text{time})$ pair are appended to the corresponding T_{node}, thereby becoming valid sources or destinations for the mapped edge sets. According to the rules defined in Eq. 2.10, the static graph is augmented with a new category of self-edges (shown in red and blue in Fig. 3.6c), which do not originate from temporal interactions but are structurally required to connect $(\text{node}, \text{time})$ pairs across splits.
Consequently, the resulting graph structure is composed of three node clusters (one per split) linked exclusively through these cross-split self-edges, which remain independent from the original static triplet (or quartet) mapped static edges $(e_{\text{event}}, e_{\text{self,src}}, e_{\text{self,dst}})$ [2]. For clarity, Fig. 3.6 provides a small example illustrating this remapping approach.



(a) Temporal graph



(b) Modified Static Expansion repr.



(c) Observation Layers approach on the modified static expansion representation

Figure 3.6: Example of the Observation Layer approach variant to a Temporal graph.

A key aspect of this design is the reduction in the number of nodes within each observation layer: only nodes actually participating in temporal edges within a given split are included to the analysis step at observation time.

It should also be emphasized once more the observation layer does not represent an actual physical observation, but rather serves as an artifact used by the model to properly cluster temporal splits. Without this mechanism, the model could gain prior knowledge of which nodes interact during validation and testing, potentially leading to data leakage. However, as explained

[2]We applied a lighter notation replacing source and destination with src and dst respectively.

in this approach and in the one described in paragraph 3.3.1, this scenario should never occur for our frameworks.

As anticipated, this revised framework resolves the major issue identified at the end of paragraph 3.3.1: the substantial removal and potential depletion of self-edges from the training and validation splits that was previously required to prevent data leakage arising from cross-split edges. Under this design, the concern is structurally eliminated: Each train, validation, and test split is augmented with an additional timestamp associated with its corresponding observation layer, which enforces strict isolation between splits while enabling all previously moved self-edges to be safely reintegrated into their original (source) split without risk. To close any remaining sources of criticality, the introduction of cross self-edges ensures the information accumulated up to a given split can be reliably propagated to the next one, preserving the continuity of node identity across time. As a result, the model retains a well-clustered temporal structure while fully maintaining the separation required to avoid leakage.

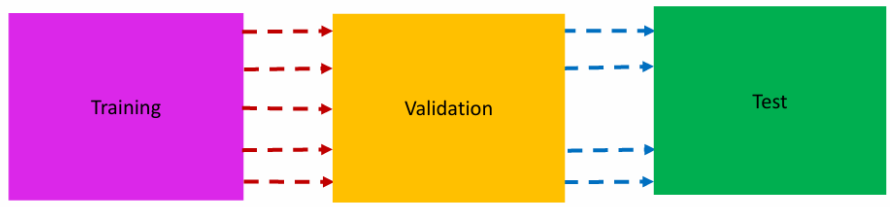


Figure 3.7: Cluster structure of the observation layers implementation.

To conclude this section, we first note the additional observation layers are also employed during the creation of negative edges, as they are fully integrated into the originally proposed scheme. Second, the only notable drawback is the further increase in node tuples, which could exacerbate memory requirements on top of the already substantial sets of defined nodes.

3.4 Data Translation - IDs mapping

We have reached the final step before applying our transformation in an experimental setting, which requires a brief remark regarding the representation of nodes in the "expanded" version. Since the nodes of the static representation are defined as tuples $(node, time)$, this format is not directly compatible with standard GNN architectures. To address this issue, we assign a unique identifier to each tuple, effectively mapping the temporal nodes back into a

conventional setting with standard $(source, destination)$ edges. In particular, we maintain separate ID spaces for source and destination nodes. It is also important to restate negative edges are constructed using only nodes that already appear in positive interactions, and therefore require no additional IDs.

The translation we apply results in a new graph representation in which the explicit temporal component is no longer present (i.e. the notion of the individual $t \in T$ is lost), but the underlying temporal ordering is safely embedded within the topology and propagated through distinct node and edge types. Consequently, the resulting structure is a heterograph partitioned into training, validation, and test sets. We now describe how the remaining components required for the model to function effectively were prepared.

Node Mappings

As previously discussed, under the hypothesis that we are working with a bipartite graph, we establish two separate mappings to convert each temporal node (i, t) into a type-specific node representation, depending on whether i corresponds to an source or a destination node:

$$
\begin{aligned}
&(source_i, t_i) \xrightarrow{\text{map}} \mathbf{src}_{i,t_i} && \text{(Source nodes), denoted as } \mathcal{V}_{\text{src}} \\
&(destination_j, t_j) \xrightarrow{\text{map}} \mathbf{dst}_{j,t_j} && \text{(Destination nodes), denoted as } \mathcal{V}_{\text{dst}}
\end{aligned}
$$

Edge Mappings

We define three distinct edge relations in the static graph, each identified by a symbolic type:

$$
\underbrace{[(source_i, t_i), (destination_j, t_j)]}_{event} \xrightarrow{map} (\mathbf{src}_{i,t_i}, \mathbf{dst}_{j,t_j}) \in \mathcal{E}_{event}
$$

$$
\underbrace{[(source_i, t_i), (source_i, t_i + \delta t_i)]}_{self,src} \xrightarrow{map} (\mathbf{src}_{i,t_i}, \mathbf{src}_{i,t_i+\delta t_i}) \in \mathcal{E}_{self,src}
$$

$$
\underbrace{[(destination_j, t_j), (destination_j, t_j + \delta t_j)]}_{self,dst} \xrightarrow{\text{map}} (\mathbf{dst}_{j,t_j}, \mathbf{dst}_{j,t_j+\delta t_j}) \in \mathcal{E}_{self,dst}
$$

These three edge types represent, respectively, the event (source– destination interactions), the source self-temporal edges, and the destination

self-temporal edges. The temporal consistency between the various pairs of t_i and t_j follows the definitions introduced in equations (2.7), (2.8) and (3.1).

In addition to the previously defined edge sets, we introduce a reversed version of the event relation. This is required to support the link prediction task, as message-passing must be able to propagate in both directions between sources and destinations:

$$\underbrace{[(destination_j, t_j), (source_i, t_i)]}_{rev\_event} \xrightarrow{map} (\mathbf{dst}_{j,t_j}, \mathbf{src}_{i,t_i}) \in \mathcal{E}_{rev\_event}.$$

Chapter 4

Experimental setting

In the following chapter, we provide a brief introduction to the Temporal Graph Benchmark (TGB) and the specific Wikipedia dataset, which will serve as the primary trial dataset for our experiments, along with all relevant details and adjustments required to work with it.

We then proceed with a detailed presentation of the various implementations we tested, highlighting individually the differences and the key aspects that motivated each approach.

4.1 Temporal Graph Benchmark

The Temporal Graph Benchmark (TGB) is an organized collection of different and challenging datasets designed for realistic, reproducible, and robust evaluation of machine learning methods on temporal graphs. It supports both dynamic link prediction and node property prediction tasks, with an integrated pipeline for dataset downloading, loading, evaluation, and leaderboard submission. Its update 2.0 further expands the benchmark by adding new datasets for temporal knowledge graphs and temporal heterogeneous graphs [20]. Among these, we find Wikipedia, which we selected as trial dataset [27].

4.1.1 Wikipedia Dataset

Belonging to TGB, this dataset represents a co-editing network on Wikipedia over the span of one month. It is modeled as a bipartite temporal interaction graph, where nodes correspond to editors and wikipages, and each temporal edge indicates that a given editor modified a specific page at a particular

timestamp. Each edge is enriched with textual features derived from the corresponding page edits.

The objective is to predict which wikipage a user will interact with at a given time, framing the problem as a temporal link prediction task (see section 2)[21].

We can define the dataset with a corresponding temporal graph

$$G = (V, E, T), \quad V_{editors},\, V_{wikipages} \in V,$$

$$V = V_{editors} \cup V_{\text{wikipages}}, \quad V_{\text{editors}} \cap V_{\text{wikipages}} = \emptyset,$$

$$E = \{\, e = (u, v, t) \mid t \in T,\, u \in V_{\text{editors}},\, v \in V_{\text{wikipages}} \,\}.$$

Note that later in our discussion, we will implicitly use the slightly modified notation $E_x = \{(u, v, t, x_{uvt}) \mid u, v \in V,\ t \in T,\ x_{uvt} \in \mathbb{R}^d\}$, which accounts for the presence of edge features that were excluded from our initial analysis, as their specific relevance to the edits was unclear. We adopted this modification in an attempt to implement approaches similar to some other models, and we will discuss their utility in our approach [39, 49].

The dataset presents 8,227 editors and 1,000 wikipages, each of them connected with a total of 157,474 temporal edges. Each temporal edges has a feature vector of size 172. It is possible to have more than one editor editing the same wikipage at the same timestamp or more wikipages edited at the same timestamp by one author (even though these cases are very rare).

From the supervised setting perspective, the dataset provides a predefined temporal split consisting of 110,232 training edges, 23,621 validation edges, and 23,621 test edges, following the TGB methodology in which all datasets are chronologically divided into training, validation, and test sets with 70%, 15%, and 15% of the edges respectively [44].

Regarding negative edges, the TGB framework provides negative sampling and loader functions. Specifically, for Wikipedia during the validation and test phases, negative edges are made available as vectors of vectors, with each positive edge associated with a list of all possible negative destinations. The size of these vectors is almost always 999, with only four exceptions (two in the validation set and two in the test set), where they contain 998 elements.

This occurs because any destination targeted by one or more temporal edges at a given timestamp is excluded from the corresponding negative list. Consequently, in the few cases where a source participates in two temporal edges at the same timestamp, the vector contains one fewer element.

Since negative training edges are not provided, they must be constructed from scratch. In general, their number can vary and is denoted as $n_{samples}$, a

value chosen to be as comparable as possible to those in the validation and test sets. In line with the values provided above we select $n_{samples} = 999$ per positive edge.

We have all that is required: the temporal graph will be mapped into one (or both) of the static implementations defined in Sections 3.3.1 and 3.3.2, along with its negative edges as described in Section 3.2, and then fed into each implementation.

From the perspective of node embeddings, we rely on three components: the projected event features that we name (X_E), the initial node features (X_I), implemented in two variants, and the output embeddings produced by the graph neural network (X_G). More precisely:

- X_E / X_{event}: The event embeddings obtained by projecting the feature vectors associated with event edges. These features, provided by the Wikipedia dataset (with 172 dimensions), encode various operations performed during each temporal edge.

- X_G / X_{GNN}: The embeddings produced by the GNN after the message-passing operation. They are obtained by feeding X_I into the GNN (see Appendix C.1);

- X_I / X_{init}: We evaluated two approaches for defining the initial node features. The first uses learnable embeddings for each node, definition applied to the implementations in Sections 4.2, 4.3, and 4.5. The second and more stable alternative assigns instead a group embedding to each chain of self-edges, as detailed later in Section 4.6.

  Both approaches were tested with two initialization strategies:

  (i) random initialization from a uniform distribution (experimented with in all four cases, but later abandoned in the final two implementations);

  (ii) a learned default feature vector, computed from the average of the warm (i.e. trained) embeddings and perturbed with Gaussian noise (the definitive strategy from paragraph 4.5 onward).

Concluding this preceding introductory paragraph, we reserve the description of specific functions and architectures for their respective sections, while the more general ones are cited in the main sections but their details are reserved for Appendix C when applied.

4.2 First Approach: Full HeteroSAGE without Node Embedding Copies

The first implementations tried are also the most direct ones: starting from the static representations described in paragraph 3.3.1 and 3.3.2, we applied a 2-layers HeteroSAGE (see Appendix C.1), where in detail we have alternated twice four SAGEConv wrapped into an HeteroConv layer and BathNorm1 layer (see Section 2 and Appendix C.2).
In this way, we have structurally utilized all the information available from our model for the message aggregation step. The situation is different with respect to the initial embeddings: since the nature of the features provided by Wikipedia for each temporal edge was not entirely clear, we initially refrained from implementing them. Subsequently, we incorporated these features as attributes for the event edges, representing the closest translation of the original graph and forming the core of our prediction task (see Section 4.1.1). For the nodes we implemented learnable embeddings X_I on which the GNN operates, as no additional data were available for them.
As explained in paragraph 3.3.1 our first step, after the removal of the redundant edges derived from the static graph construction (see Appendix A), is the moving of the cross self-edges into their destination split. After assigning learnable embeddings to the nodes, the training-validation-test cycle begins, carefully ensuring that the correct set of edges is used at each step. To facilitate this, we have defined three subgraphs: data_train, data_val, and data_test (although the last one is not strictly necessary).

During the training and validation phases, only edges in the training set are visible to the GNN, whereas at test time the GNN has access to both the training and validation edges. At the end of each training iteration, the model computes the loss (see Appendix C.4) using the scores produced by the MLP decoder (see Appendix C.3) and updates the node embeddings via backpropagation. Model performance is then evaluated using the standard Mean Reciprocal Rank (MRR) metric (see Appendix C.5), following a standard pipeline.

4.3 Second Approach: HeteroSAGE with Node Embedding copies

This approach was developed to address some critical issues observed in the previous model (see Paragraph 5.1). Here, we rely only on the lighter static mapping described in Section 3.3.1 to construct the graph. We then ex-

plored two message aggregation strategies. The first approach propagates messages exclusively along event edges, enforcing a more conservative assumption about information flow, even though this may lead to less effective information retrieval (see Section 5.2). The second restores the original GraphSAGE architecture, allowing aggregation from the full neighborhood as in the previous implementation (see Appendix C.1).

It is important to note that self-edges were maintained in both configurations: beyond their role in message aggregation, they serve a distinct and essential function by enabling the temporal propagation of node embeddings across the system. Prior to presenting the model, we provide a brief clarification of this copy propagation mechanism.

4.3.1 Copy procedure

The copy procedure stems from the idea that chains of self-edges represent the temporal evolution of a single node in the original temporal graph (see Section 2.1.2). From this perspective, if a node along such a chain neither interacts with external nodes nor undergoes any endogenous change, then it should share initially the same embedding with all the other nodes of the chain.

Motivated by this observation, the copy procedure addresses the "cold" initialization problem by propagating, after each training step, the embeddings of the learned latest temporal nodes (i.e. those with the highest timestamp in training after the mapping step) along their corresponding temporal-evolution chains into the subsequent validation and test splits. The underlying principle is that validation and test nodes should inherit the final training state as their initial state, since they represent the same original temporal node.

In practice, this procedure is executed in two main stages. The first stage, itself divided into two steps, begins by initializing all nodes with learnable embeddings while immediately zeroing the embeddings of those belonging to the test split. Subsequently, we apply the functions which identify the connected components (i.e. the chains) defined by the test and cross-to-test self-edges.

Due to the relocation of cross-split self-edges (see Section 3.3.1), each chain is expected to contain exactly one node originating from the training or validation split, and thus carrying a non-zero embedding. This node is designated as the copy source and is stored in a dictionary mapping every node in the chain to the corresponding "warm" node ID to be used later in the copying mechanism.

The second part of this phase follows the same logic but is applied to the

validation nodes: their embeddings are zeroed, the clusters are re-identified but in this case for the validation split, and their respective warm sources are determined in the same manner.

The second main stage takes place at the end of each training step, when the actual copy operation is executed. Finally, during the test phase, the optimal embeddings for both the training and validation nodes are transferred to the test nodes before to perform the final MRR evaluation.

Having described the copy procedure, we can now conclude the initial variable definitions: no edge features were incorporated on the event edges, meaning that the model relied entirely on learnable node embeddings.

These clarifications allow us to resume the previous discussion, with the understanding that our approach evolves along two parallel lines. On one side, the event edges alone in the raw configuration, and subsequently both event and self-edges in the effective configuration, are processed by the GNN and the MLP to update the node embeddings and perform the prediction task. On the other side, the self-edges propagate these updated embeddings independently across time into the other partition of the system, theoretically providing improved initial embeddings[1] for all nodes, which would otherwise remain entirely cold-initialized.

4.4 Revisiting Node Embeddings

Investigating the results of the two previous implementations (Sections 5.1 and 5.2) we highlight a significant limitation: the lack of sufficiently specific initial features and embeddings, which we refer to as **Criticality 0: Insufficient Node Representations**. In the previous approaches, we attempted to progressively enrich the graph with all available components, but this lead to only marginal improvements.

Recalling the concluding discussion in Section 3.3.1, it is unsurprising that no substantial performance gains were observed. Upon closer analysis, two additional issues emerge that further limit the achievable improvements:

- **Criticality 1: Self-Edge depletion.** Table 4.1, which reports the number of unique sources $(editor, t)$ and unique destinations $(wikipage, t)$ of the static graph representation, reveals an important observation: almost all nodes appear only once within each temporal split. This implies the self-edge depletion problem is already fully in effect during the training phase when using the lighter static representation (see

[1] We will discuss why this is not strictly true in the third implementation.

Section 3.3.1), as very few self-edges are available for the message aggregation step.

	Total temp./event edges	Unique sources	Unique destinations
Training	110,232	110,222	110,226
Validation	23,621	23,619	23,621
Test	23,621	23,619	23,621

Table 4.1: Summary of original temporal/event edges and unique nodes in the training, validation, and test splits of the Wikipedia dataset. Sources correspond to $(editor, t)$ nodes and destinations correspond to $(wikipage, t)$ nodes.

- **Criticality 2: "Fully-Cold" Nodes in Validation and Test Splits.** The second criticality arises from nodes that appear directly in the validation or test splits without any representation in the training set. As observed in Figure 4.1, approximately 20.7% of editor nodes and 4.8% of wikipage nodes from the original temporal graph are missing from the training set. Consequently these nodes, or more precisely their $(node, timestamp)$ pairs, lack a "warm" starting embedding derived by the training phase from which information can propagate. They are therefore subject to complete cold initialization, which must be explicitly addressed in the design of subsequent models.

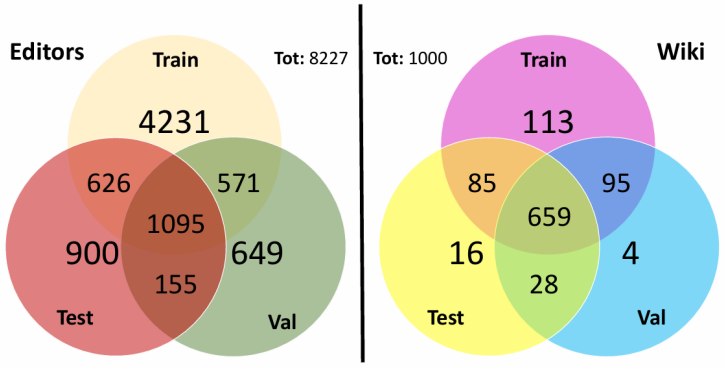


Figure 4.1: Number of editor and wikipage nodes present in each split of the temporal graph representation. Note that 1,704 editor nodes and 48 wikipage nodes are not present in the training set.

Motivated by these observations, the two following implementations operate on the static representation described in Section 3.3.2 employ an improved

copy procedure, and utilize better-structured node embeddings, as outlined below.

4.5 Third Approach: HeteroSAGE with corrected Node Embedding copies

In this implementation, we aim to address the criticalities described above as effectively as possible. While the model is not expected to differ substantially from the implementations discussed in Approaches 4.2 and 4.3, it is modified wherever necessary to achieve the various objectives. As motivated in Section 3.1, we adopt the Observation Layers approach for the static graph representation. In this way, at the cost of a slightly higher memory requirement, Criticality 1 (4.4) is structurally eliminated. The remaining components are largely unchanged from previous implementations: we utilize a HeteroSAGE (see Appendix C.1) operating on both event and self-edges, an MLP decoder (see Appendix C.3), and integrate the available features associated with the event edges, as in Approach 4.3.

The core change which deviates from the previous implementation and truly characterizes it is the introduction of a corrected copy mechanism, which now takes into account the Criticality 2 (see 4.4) and poses an end to it. To reach this goal the new mechanism performs a standard copy operation whenever valid warm references exist, while assigning a learnable fallback embedding to nodes that lack any such references (i.e. the "fully cold" starting nodes). This fallback value is computed from the mean embedding of the warm nodes, which is treated as a learnable parameter, passed through a linear and non-linear transformation, and finally perturbed with Gaussian noise (see Section 4.5.1).

Before proceeding, it is necessary to explain why this learnable fallback approach was chosen as our definitive solution. Prior to this, we explored an intermediate strategy between the two methods cited at the beginning of this chapter (4.1.1) for defining X_I. Instead of assigning fully random values, we computed two separate averages of all embeddings belonging to warm nodes: one for editor nodes and one for wiki page nodes. These averaged embeddings values were then used to initialize the corresponding cold-start nodes, depending on whether they originated from editors or wikipages. The intuition is straightforward: even without direct historical information for these nodes, it is reasonable to assume that they are broadly similar to the overall population within their category. Initializing them with a representative embedding should therefore provide a more meaningful starting point than

purely random values. This approach accomplished modest improvements, in part because it still lacked a crucial element: variability.

To address this, we introduced the aforementioned final refinement through the **Learnable Inductive Fallback** (Sec. 4.5.1). In this method, the average embedding is promoted to a learnable parameter, allowing it to adapt during training and better align with the downstream task. Additionally during evaluation, a small Gaussian noise is injected to ensure that each node remains distinguishable, preserving node-specific variability despite the absence of prior interactions. Overall, this refinement enhances inductive generalization: full-cold nodes maintain a meaningful connection to the trained embedding space while still being able to represent unique temporal trajectories and structural roles.
To summarize, in the definitive version we:

- Apply the copy process described in Section 4.3.1, using the corrected version detailed in Section 4.5.1, to all nodes that have at least one timestamp in the training split.
- For the remaining "full-cold" nodes (editors and wikipages isolated from the training phase), initialize their embeddings using the fallback value computed from the warm nodes, and inject a small Gaussian noise component to preserve variability across the "fully-cold" chains.

It has not been explicitly stated before but the initialized X_{init} are learnable embeddings as are the X_{GNN} embeddings. It is important to note however that the event features themselves are not trainable because they are fixed and provided by the original dataset while their projected embeddings X_{event} can be updated via the backpropagation process.

With our embeddings now formalized, we feed the initialized embeddings X_{init} into the GNN, following the same procedure as in the previous versions.

We reserve this final space to define more precisely how Learnable Inductive Fallback mechanism works.

4.5.1 Learnable Inductive Fallback Mechanism

We now describe this more refined mechanism, which has proven to be more effective in our studies.

The **Learnable Inductive Fallback** module defines a trainable default feature vector for isolated nodes (i.e. fully cold-start nodes with no connections to the training phase through self-edges).

This module takes as input a d-dimensional vector representing the global average of all warm self-edge chain embeddings [2]. As first step, this average is converted into a learnable parameter denoted as $\mathbf{f}_{mean}$, which can be updated via backpropagation during decoder training. Then, a linear transformation is applied:

$$\mathbf{f}_{lin} = \mathbf{W}_{linear} \cdot \mathbf{f}_{mean} + \mathbf{b}_{linear}, \tag{4.1}$$

followed by a nonlinear activation

$$\mathbf{f}_{fallback} = \tanh(\mathbf{f}_{lin}), \tag{4.2}$$

which stabilizes the learned feature values within $(-1, 1)$, ensuring compatibility with the decoder (MLP) feature space.

During training, $\mathbf{f}_{fallback}$ is used to directly initialize cold nodes. During evaluation, a small Gaussian perturbation is added to improve robustness:

$$\mathbf{f}'_{fallback} = \mathbf{f}_{fallback} + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2_{noise}). \tag{4.3}$$

As explained, this inductive regularization introduces controlled variability into the initialization of cold nodes, enhancing the model's generalization capability.

4.6 Fourth Approach: HeteroSAGE with group Node Embedding

This approach arises from a necessary shift in approach: previously each node has been assigned a distinct individual embedding, treating nodes as isolated building blocks from which temporal trajectories are formed (i.e. the chains constructed via self-edges). However, this view may diverge significantly from the intended notion of a persistent, temporally evolving "unique" node identity, which we are already exploiting through the copy process (see 4.3 and 4.5).

To better preserve identity continuity, this implementation introduces group embeddings, where a single embedding represents all nodes within a complete chain of self-edges (i.e. the temporal evolution of a single node). In this way, the representation captures the intrinsic identity of a node over time, rather than being limited to individual temporal snapshots.

Accordingly, to mitigate the "cold-start" problem for groups unseen during training (i.e. the nodes recorded in Fig. 4.1), we extend the Learnable Inductive Fallback module (Section 4.5.1) to operate at the group level.

[2] Recall we distinguish between the average values for editor nodes and wiki page nodes.

Specifically during training, the embeddings of inactive (cold) groups are replaced with the learned fallback vector, which is initialized from the average embedding of all active groups and perturbed with a small Gaussian noise to preserve expressiveness and prevent convergence toward indistinguishable representations. In practice, this requires only a modification of the module's input.

For completeness, we briefly describe the process used to define the groups (i.e. the chains representing the unique identity of the original temporal nodes). During the construction of the positive mapping (see Appendix A), we record the original node $\in \{editor, wikipage\}$ corresponding to each self-edge. Using this information, we construct a dictionary in which each key is a tuple $(node, timestamp)$, mapped to the unique identifier defined in Section 3.4, and the corresponding value of the dictionary represents the original temporal node. This mapping allows us to explicitly define groups of nodes that correspond to the same identity entity, ensuring that each group shares a single learnable group embedding.

The encoder employed in this implementation remains the full HeteroSAGE architecture, while the MLP decoder is used as described in Appendix C.3. The three types of embeddings, $X_{init,group}$, X_{event}, and X_{GNN}, are concatenated to form an enriched representation, a choice intended in part to mitigate Criticality 0 (see Approach 4.4), which is subsequently passed to the decoder.

4.7 Node Embeddings Ablation

This second pause in the discussion of the approaches arrives because, despite extensive iterative refinements and a preliminary fine-tuning phase, all previous models reached a level of stability without achieving comparably valuable improvements (Chapter 5). This suggests that we have likely reached the limit of what can be achieved within the current framework, or that we may have overlooked something along the way. A straightforward way to begin verifying one or both of these hypotheses is to perform an ablation study, testing various combinations of embeddings fed to the MLP decoder. In particular, since X_{GNN} and X_{event} are both associated with two key questions, this analysis can provide insight into their individual contributions:

- Whether the chosen encoder is appropriate (see Appendix C.1);
- Whether the provided features are effectively exploited (see Paragraph 4.1.1).

As evidenced by the results shown in Table 4.2, the models including X_{GNN} embeddings exhibit the most severe degradation, while those incorporating X_E also decline, though far more mildly. In contrast, the X_I embeddings provide a reasonable baseline, clearly not optimal, yet with evident room for improvement[3].

Concatenation	Test_MRR	Val_MRR
XI	**0.279 ± 0.013**	**0.339 ± 0.007**
XI_XE	0.260 ± 0.014	0.320 ± 0.017
XGNN_XI	0.014 ± 0.002	0.017 ± 0.001
XGNN_XE	0.013 ± 0.002	0.014 ± 0.001
XGNN	0.012 ± 0.002	0.022 ± 0.002
XGNN_XE_XI	0.012 ± 0.003	0.012 ± 0.002
XE	0.005 ± 0.000	0.003 ± 0.000

Table 4.2: Comparison of the different concatenation configurations, presented in descending order of performance.

Based on these results, the core issue appears to lie within the GNN architecture itself, which struggles to extract meaningful information from the highly sparse graph structure and the functional but not fully optimal node embeddings. This limitation ultimately represents the primary source of performance degradation, rather than the quality of the initial node features X_I which are learnable embeddings initialized from a uniform random distribution and progressively refined according to the implementation's update rules.

This outcome is not entirely unexpected. The use of classical GNNs in temporal graph settings (i.e. without explicitly discarding the temporal component) remains relatively unexplored, whereas successful approaches typically rely on carefully designed Temporal GNN mechanisms. Notable examples include the foundational TGN model [39] and the more recent TPNet model [30], which currently represents the state of the art on the Wikipedia leaderboard[4].

We can also argue that both our static mapping procedures 3.3.1 and 3.3.2 further amplify graph sparsity due to the small size of Wikipedia: each original node is expanded into multiple timestamped tuples. Although these

[3]Note that in this case, the results were obtained using the best hyperparameters from the subsequent Baseline 4.8, due to time constraints preventing additional fine-tuning. In any case, the results clearly reveal the underlying issues.

[4]Reported according to the most recent leaderboard data November, 2025.

tuples are linked through self-edges, such connections could be too weak to counteract the resulting fragmentation, making it even more difficult for the model to aggregate the already limited information into meaningful global representations.

Building on this analysis, we conclude the best-performing configuration requires a reconsideration of both the encoder and the features, in order to potentially recover the model from a suboptimal solution. Addressing this issue in depth would require a more extensive analysis, which could not be fully addressed within the timeframe of this work and is therefore left for future studies.

Instead, at the conclusion of this chapter, we present a baseline strategy, which will constitute a starting point that the aforementioned encoder should surpass in order to be considered effective.

4.8 Baseline Approach: Decoder-Only Model

Following the insights gained in the previous paragraph, we present the baseline approach, intended as a benchmark for evaluating future encoder configurations. Given that the encoder itself requires further investigation, this implementation adopts a counterintuitive approach: a fully decoder-based model. While this concept may initially seem misleading, it provides the most suitable reference for comparison, as it entirely omits the encoder, which is the core component to be addressed in future work.
Furthermore, we remove X_{event}, originally included as features derived from the temporal graph (see Section 4.1.1), since, as shown in Table 4.2, they are not particularly effective. This choice also leaves the door open to exploring more informative features, such as those that better capture the underlying graph topology or the time differences implicitly lost during the IDs mapping phase (see Paragraph 3.4), which will be addressed in future studies.

Therefore, the described strategy relies solely on the group embeddings, $X_{I,group}$, which are processed according to the Learnable Fallback Module and the copy procedure to handle cold nodes, then fed to the MLP decoder (see Appendix C.3) and adjusted exclusively through loss evaluation and backpropagation.

It is worth noting the graph structure remains relevant, albeit to a limited extent, since the group embeddings are defined based on the self-edges. As an incidental advantage, this architecture can be applied to both static representations of the graph (see Sections 3.3.1 and 3.3.2), as there should be no significant difference between them, given that the graph topology is

not directly utilized.

Before moving to the next chapter, it is worth noting this architectural decision aligns with the prior cited work applying static expansion to temporal graphs. In particular, both Sato et al. [42] and Piaggesi and Panisson [37] rely on embedding techniques grounded in standard neural network architectures, random-walk–based node embeddings in the former [36, 13], and higher-order skip-gram with negative sampling (SGNS) [33] in the latter, closely related to MLP-style learning rather than GNN message-passing. This connection reinforces the suitability of our decoder-only formulation within the broader literature on temporal graph embedding methods. At the same time, our approach provides the opportunity for more graph-aware implementations, where incorporating both the graph topology and the history of node interactions, as demonstrated in Lu et al. [30], has been shown to be essential for accurate link prediction in temporal graphs.

Chapter 5

Results

In this chapter, we present the results obtained from five runs of each implementation. We first provide a table summarizing the different variants considered, based on the methods described in Chapter 3, followed by a comprehensive table of overall results. Within each section, we report the specific results obtained and provide any additional details necessary to support the accompanying discussion and analysis. A summary table of the hyperparameters tested, including the values explored prior to selecting the final configurations, is provided instead at the end of this chapter.

References and Global Results

ID	Approaches	Static representation
00_Impl	Full HeteroSAGE without NE copies	Data Splits
01_Impl		Observation Layers
02_Impl	Event only HeteroSAGE with NE copies	Data Splits
03_Impl	Full HeteroSAGE with NE copies	
04_Impl	Full HeteroSAGE with correct NE copies	Observation Layers
05_Impl	Full HeteroSAGE group NE	
06_Impl	Decoder – ONLY	Data Splits
07_Impl		Observation Layers

Table 5.1: Approaches Overview.

ID	Val_MRR	Test_MRR
00_Impl	0.007 ± 0.001	0.007 ± 0.002
01_Impl	0.008 ± 0.000	0.008 ± 0.000
02_Impl	0.400 ± 0.199	0.047 ± 0.024
03_Impl	0.706 ± 0.362	0.612 ± 0.475
04_Impl	0.010 ± 0.002	0.009 ± 0.003
05_Impl	0.012 ± 0.002	0.012 ± 0.003
06_Impl	0.516 ± 0.000	0.468 ± 0.002
07_Impl	**0.519 ± 0.003**	**0.470 ± 0.004**

Table 5.2: Global results of all the implementations. The best result is underlined and in bold, the second is only underlined.

5.1 First Approach: Full HeteroSAGE without Node Embedding Copies

ID	Approach	Static representation
00_Impl	Full HeteroSAGE without NE copies	Data Splits
01_Impl		Observation Layers

Table 5.3: Variants of the First Approach.

ID	Val_MRR	Test_MRR
00_Impl	0.007 ± 0.001	0.007 ± 0.002
01_Impl	0.008 ± 0.000	0.008 ± 0.000

Table 5.4: Results of the variants of the First Approach.

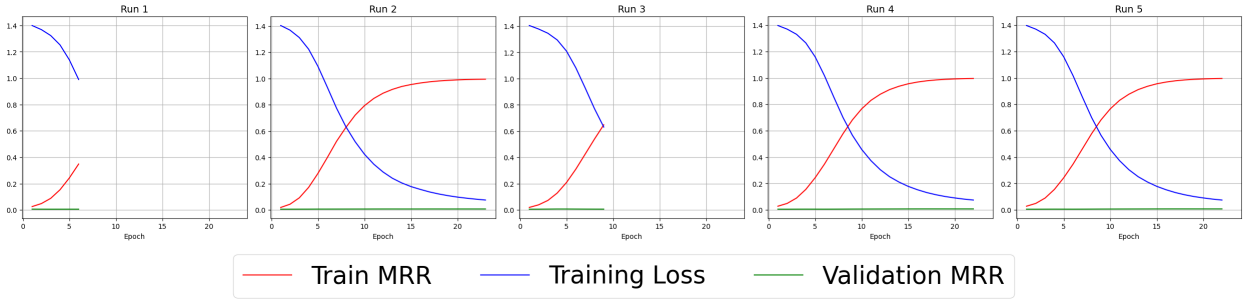


Figure 5.1: First Implementation (00): Plot of the 5 runs. Analogous results are given by the second Implementation (01).

The first approach described in Paragraph 4.2 did not lead to good results for either version of the static representation, which is not unexpected considering that our initial attempt was fundamentally preliminary. A closer look at the results of each individual run for the first implementation (00) (Figure 5.1) [1], suggests that the model is overfitting. Considering these observations, we can begin speculating about possible causes of this behavior. We briefly outline some of them and will attempt to adjust our approach accordingly at a later stage:

- The graph is very sparse: each node typically has three edges, one event edge and two self-edges. Exceptions include the temporally starting and ending nodes (those with minimal or maximal timestamps), which have only one or two edges (self only, or self and event), as well as the rare cases involving multiple events. This pattern can be readily inferred from Table 4.1.

- The graph lacks sufficiently informative embeddings. Since our approach relies almost exclusively on the static graph structure, which is sparse and provides limited relational signal, the learnable node embeddings alone may not effectively capture all relevant dependencies. Furthermore, enriching the model with event-level features and their corresponding embeddings does not appear to bring a measurable performance improvement. This outcome is predictable, as these event features do not seem to introduce any meaningful additional information to the model, as later demonstrated in Table 4.2;

- The graph contains a significant number of "cold" nodes. As mentioned in Appendix B.2, not all editors and wiki pages tuples are included in

[1]The second implementation (01) shows analogous results and is therefore not reported

the training phase. Consequently, all their representations miss an essential refinement phase.

5.2 Second Approach: HeteroSAGE with Node Embedding copies

ID	Approach	Static representation
02_Impl	Event only HeteroSAGE with NE copies	Data Splits
03_Impl	Full HeteroSAGE with NE copies	

Table 5.5: Variants of the Second Approach.

ID	Val_MRR	Test_MRR
02_Impl	0.400 ± 0.199	0.047 ± 0.024
03_Impl	0.706 ± 0.362	0.612 ± 0.475

Table 5.6: Results the variants of the Second Approach.

This second version, described in 4.3, confirms a full HeteroSAGE is the most promising approach, as it can act on the full graph structure and does not appear to exhibit any evident signs of data leakage. In both cases, the copy procedure described in 4.3.1 seems to be a reasonable way to mitigate overfitting. However, the considerable variance and general instability of the model indicate that some important aspects are still missing from our approach. After a brief analysis, several issues become immediately apparent:

- As explained in Section 4.4 and previously noted in paragraph 5.1, a significant portion of nodes are "cold," meaning they are absent from the learning process, even though paths of self-edges connect them to it. Another case, representing our central challenge, involves nodes that have no connections to the training set. Specifically, as shown in Figure 4.1, approximately 20.7% of editor nodes and 4.8% of wiki page nodes from the original temporal graph are completely absent from the training set. Consequently, these nodes, or more precisely their associated $(node, timestamp)$ pairs, will never have a "warm" starting node from which information can be copied. They are therefore subject to critical zero initialization, which we identify as a primary cause of

the high instability observed, and which must be explicitly addressed in future modeling;

- Continuing along the line of embedding-related issues, we note once more the imposed values for warm nodes may also not be particularly effective. They are merely random uniformly distributed embeddings which, although partially updated during training, certainly do not provide an optimal starting point;
- Finally this point was noted previously, and we now provide a more detailed explanation. A confirmed limitation, which we will quickly address moving forward, stems from the removal of self-edges in the initial message aggregation attempt (i.e. 02_*Impl*). This decision leads to a substantial loss of information regarding node identity and temporal continuity, which is only weakly preserved through the copy process. As a result, the graph presented to the GNN consists solely of node-to-node event links, making the structure highly sparse, minimally informative, and prone to unstable learning dynamics.

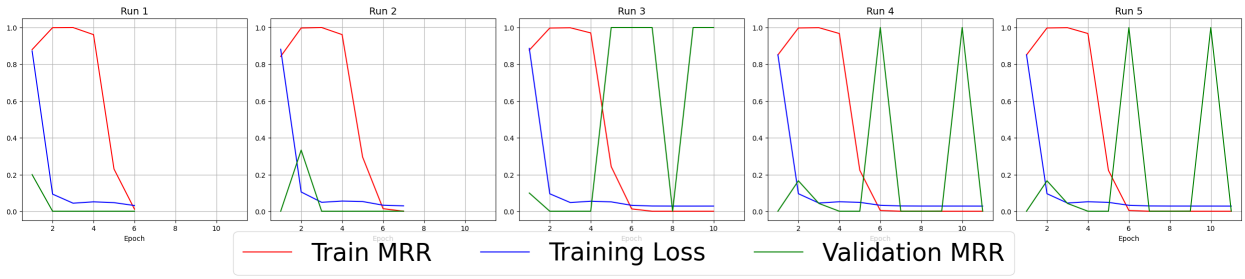


Figure 5.2: Second Implementation (03): Plot of the 5 runs.

The results in Table 5.6 and the Val MRR outcomes, which could reveal the persistence of overfitting despite fluctuations (Fig. 5.2), indicate that high variability alone does not account for all the observed issues. The results presented correspond to the model incorporating self-edges in message aggregation; nevertheless, their inclusion does not appear to provide a clear performance improvement. The underlying issue may not be whether self-edges participate in message-passing, but if they are actually included in the learning split.
This observation is not casual but follows from a careful analysis of the original temporal edges. Within the training split (and similarly in the validation and test splits), almost every node appears exactly once per temporal segment and consequently participates in a single self-edge, with only negligible exceptions (Table 4.1).

Because of the way the Data splits representation is defined (Paragraph 3.3.1), self-edges are effectively reallocated and do not appear frequently enough during training for the model to exploit the temporal continuity they are intended to capture (as discussed in Paragraph 4.4). As a result, they rarely contribute meaningfully to the learning process.

This finding is crucial for two reasons: first, node-specific information is extremely sparse; second, due to the temporal splitting strategy (see 3.3.1), many self-edges fall outside the training segment, further reducing their utility. Consequently, the model is unable to employ self-edges effectively as part of the representation learning process.

To recover the limited information expressed by self-edges while preventing data leakage, it is necessary to substantially revise the static graph structure. At the same time, the model should maximize information propagation so that each node can acquire as much meaningful context as possible without introducing additional sources of instability. Guided by these requirements, we revert to the heavier Observation Layers representation, which does not exhibit Self-Edge Depletion (Sections 3.3.2, 4.4).

5.3 Third Approach: HeteroSAGE with corrected Node Embedding copies

ID	Approach	Static representation
04_Impl	Full HeteroSAGE with correct NE copies	Observation Layers

Table 5.7: Variant of the Third Approach.

ID	Val_MRR	Test_MRR
04_Impl	0.010 ± 0.002	0.009 ± 0.003

Table 5.8: Result for the variant of the Third Approach.

Thanks to the improved node initialization and updating modules for cold nodes, which now handle all possible scenarios, the model exhibits greater stability. However, the results remain insufficient and are still characterized by pronounced overfitting[2]. The main challenges that may still arise are:

[2]We do not report any training curves here, but the observed behaviors are comparable with the results shown in Figure 5.1.

- As in the First Approach (see 5.1), the graph structure may remain too sparse to effectively capture and propagate certain structural information present in the data;
- The resulting embeddings are still of limited quality: as previously discussed, the construction of X_{event}, which we attempted to reintroduce, lacks a well-defined semantic interpretation and the update mechanisms provided by the GNN and fallback module may not be as informative as desired. In addition X_I, which is uniformly generated for each individual node, lacks any notion of unique temporal node identity, a notion we know to be relevant and which has so far been transmitted via the self-edge chains and the copy mechanism.

Since this last point appears to be a relevant aspect that can be effectively addressed, we will define a group embedding for each self-edge chain, i.e. for each temporal evolution of a node in the original temporal graph (Approach 4.6).

5.4 Fourth Approach: HeteroSAGE with group Node Embedding

ID	Approach	Static representation
05_Impl	Full HeteroSAGE group NE	Observation Layers

Table 5.9: Variant of the Fourth Approach.

ID	Val_MRR	Test_MRR
05_Impl	0.012 ± 0.002	0.012 ± 0.003

Table 5.10: Result for the variant of the Fourth Approach.

In this case, it can be noted the results show some improvement although not to a degree that can be considered statistically significant. On the positive side, the model still exhibits consistent stability, but the underlying issues in the dataset and current approaches clearly remain unresolved.
What remains to be done is an ablation study to identify which element may be the critical factor and if possible, determine how to address it appropriately (see 4.7).

5.5 Baseline Approach: Decoder-Only Model

ID	Approach	Static representation
06_Impl	Decoder – ONLY	Data Splits
07_Impl		Observation Layers

Table 5.11: Variants of the Baseline Approach.

ID	Val_MRR	Test_MRR
06_Impl	0.516 ± 0.000	0.468 ± 0.002
07_Impl	0.519 ± 0.003	0.470 ± 0.004

Table 5.12: Results of the variants of the Baseline Approach.

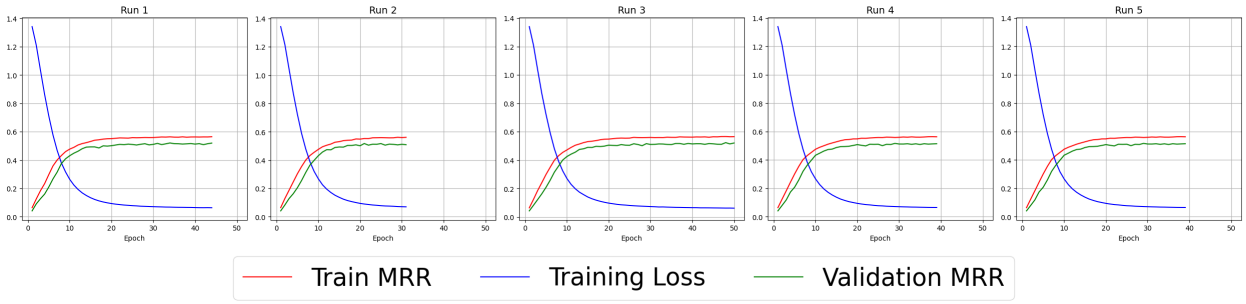


Figure 5.3: Second Implementation (07): Plot of the 5 runs.

Building on the conclusions drawn in Paragraph 4.7, and in particular on the results presented in Table 4.2, we obtain a solid starting point from which develop an actual encoder structure and to further refine the embedding design in future work (see Fig. 5.3)[3]. It is worth noting the configuration employing the Observation Layers achieves marginally better performance than the Data Splits configuration, as shown in Table 5.15.

These values alone are comparable to those of the current 11th and 12th models on the effective leaderboard for the Wikipedia dataset [45][4].

[3]Analogous results were obtained with the first implementation (06) and are therefore not reported.

[4]Results updated to November 2025.

It is unnecessary to emphasize substantial advancement remains necessary before reaching performance comparable to the top-ranked models such as TPNet [30], HyperEvent [11], and DyGFormer [54].

For clarity and ease of comparison, the previously cited results are summarized below.

Rank	Method	Test MRR	Validation MRR
1	TPNet	0.827 ± 0.001	0.842 ± 0.001
2	HyperEvent	0.810 ± 0.002	0.824 ± 0.002
3	DyGFormer	0.798 ± 0.004	0.816 ± 0.005

Table 5.13: Top-ranked results on the Wikipedia dataset leaderboard (MRR on test and validation). Source: TGB [45].

Rank	Method	Test MRR	Validation MRR
11	EdgeBank(unlimited)	0.495	0.527
12	HTGN(UTG)	0.464 ± 0.005	0.523 ± 0.005

Table 5.14: Leaderboard for the Wikipedia dataset (MRR on test and validation). Source: TGB [45].

Rank	Method	Test MRR	Validation MRR
NC	Baseline (Observation Layers)	0.470 ± 0.004	0.519 ± 0.003
NC	Baseline (Data Splits)	0.468 ± 0.002	0.516 ± 0.000

Table 5.15: Best Achieved Performance across All Experimental Configurations Evaluated in this Study.

Hyperparameters

We report here the hyperparameters tested and the corresponding best values for the Baseline Approach (Section 5.5), which underwent full fine-tuning. Implementations 06 and 07 achieved their best performance using the same hyperparameters.

Two observations are particularly noteworthy:

- The optimal batch size is 32,768, which is relatively large. This is necessary to achieve the target ratio of negatives to positives of roughly

999:1 (see Section 4.1.1). With this batch size, each batch contains approximately 33 positive edges and 32,735 negative edges. Since the actual dataset has a 1:1 ratio between positive and negative event edges (157,474 of each), we must oversample negatives or undersample positives when constructing batches. This ensures each batch maintains the desired 999:1 ratio, providing enough negative examples to give meaningful signals for the relatively small number of positive edges and to stabilize training;

- We consider two additional, correlated hyperparameters related to the number of each variant of negative edges during training (note that validation and test sets are fixed to contain only hard negatives (see Paragraph 3.2)): the number of hard negatives and the ratio between Very Weak and Weak negative variants (see Appendix B). The optimal values support our hypothesis: the achievement of strong performance does not require a large number of hard negatives. Instead, a balanced mixture of the different negative variants produces the best results, improving the model's generalization ability.

To conclude, for the remaining implementations, only limited or no fine-tuning was performed. Their results are included solely for completeness and transparency, as further optimization was deemed unnecessary given their limited reliability.

06/07_Implementation Hyperparameters		
Hyperparameters	**Best Value**	**Tried Values**
Learn Rate	3×10^{-4}	$[1 \times 10^{-5}, 2 \times 10^{-4}, 3 \times 10^{-4}, 4 \times 10^{-4}]$
Batch Size	32768	[16384, 65536]
Hidden channels	128	256
Embedding dims	64	[128, 256]
Decoder dropout	0.3	0.2
Weight decay	1×10^{-4}	$[1 \times 10^{-5}, 1 \times 10^{-4}, 2 \times 10^{-4}, 4 \times 10^{-4}]$
Gamma	0.98	[0.97, 0.99]
Num hard neg	175	[0, 50, 100, 150, 200, 500, 750, 950]
Ratio Very weak:weak neg	6:1	[1:1, 1:2, 2:1, 5:1, 7:1, 10:1]
Noise	0.02	[0.001, 0.015, 0.025]
Epochs	70	50
Warmup epochs	0.2	0.1
Patience	7	5

Table 5.16: Hyperparameter search and best values for Decoder-only Implementations (06/07).

Hyperparameters	**00_Impl**	**01_Impl**	**02_Impl**	**03_Impl**	**04_Impl**	**05_Impl**
Learn Rate	1×10^{-5}	1×10^{-5}	1×10^{-4}	1×10^{-4}	1×10^{-5}	1×10^{-5}
Batch Size	65536	65536	16384	16384	16384	16384
Hidden channels	32	32	32	32	32	64
Embedding dims	32	32	32	32	32	32
Out channels	32	32	32	32	32	32
Encoder dropout	0.5	0.5	0.6	0.5	0.2	0.2
Num_neigh	[20,10]	[20,10]	[20,10]	[20,10]	[30,15]	[30,15]
Decoder dropout	0.3	0.5	0.4	0.4	0.3	0.3
Weight decay	1×10^{-2}	1×10^{-2}	1×10^{-2}	1×10^{-2}	1×10^{-2}	1×10^{-2}
Gamma	0.99	0.99	0.99	0.99	0.99	0.99
Num hard neg	950	950	250	250	250	250
Very weak: Weak neg	1:1	1:1	1:1	1:1	1:1	1:1
Noise	None	None	None	None	0.1	0.001
Epochs	50	50	50	50	50	50
Warmup epochs	0.05	0.05	0.05	0.05	0.2	0.2
Patience	5	5	5	5	5	5

Table 5.17: Hyperparameter settings for all the implementations.

Chapter 6

Conclusion

In this Master's Thesis we investigated a static representation strategy for Temporal Graphs based on the supra-adjacency construction introduced first by Oettershagen, Kriege, Morris, and Mutzel and later refined by Sato et al. in [34, 42], extending it to a supervised learning setting, which has not yet been explored in the literature. Our analysis focused specifically on the task of Dynamic Link Prediction, although the proposed framework can be readily adapted to other tasks, which will be considered in future work.

This approach can be positioned between fully static methods, where the temporal component is entirely discarded and classical GNNs are applied to a time-agnostic version of the graph, and dynamic architectures such as TGN [39] and DyRep [49], which preserve temporal information through memory modules and other model-specific mechanisms.

In our case, temporal information is embedded directly into the topological structure of the graph, at the cost of increased memory requirements. This design choice however allows the use of standard, well-understood, and computationally efficient GNN architectures on the transformed-to-static representation.

The first stage in transforming a temporal graph into its corresponding static representation consists of mapping positive temporal edges to their static counterparts. In the context of a directed bipartite graph, each original temporal edge is translated into three distinct edges: one event edge, encoding the original interaction, and two self-edges (one for the source node and one for the destination node), which capture the temporal evolution of the corresponding nodes throughout the observed sequence of events.
Further refinements were necessary to ensure a complete and accurate translation. To this end, we introduced a final layer of nodes, referred to as the Observational Layer, which serves as a structural placeholder, thereby enabling the full transformation of each edge.

To address potential inconsistencies due to data leakage, two alternative constructions were developed, each with their own advantages and limitations: the Data Splits representation and the Observation Layers representation.

Having established the transformation of the original graph, we transposed the Dynamic Link Prediction task within the supervised setting. In this context, the problem reduces to standard Link Prediction performed on the event edges, which requires a precise definition of negative edges.

We carried out four distinct approaches, iteratively refining the strategy to address challenges such as cold-start initialization, graph sparsity, and insufficient node representations. Despite these efforts, the final implementation achieved stability but remained largely ineffective.

The closing phase involved an ablation study aimed at identifying the components responsible for the model's shortcomings, which revealed that both the encoder architecture and the employment of the original event features were the primary sources of performance degradation.

Due to time constraints, we did not implement a new encoder. Instead, we developed a baseline model to serve as a foundation for future studies. In this configuration, the pre-existing encoder was omitted, leaving only the standard MLP decoder, which produced reasonable results with potential for further optimization. These results were subsequently compared against the current Wikipedia leaderboard to measure their relative performance.

Having reached this stage, we can answer the initial research questions and present the outcomes of our study:

RQ1: Temporal Consistency. How can we construct a mapping from continuous-time temporal graphs to static representations that strictly preserves the chronological order of events, thereby preventing data leakage and ensuring the validity of the predictive model?

A1: To address this question, we build from the Supra-Adjacency methodology [34], later refined in [42]. We first add a layer of nodes at the end of the series of events to achieve a complete transformation of all temporal edges. This thus ensures by construction that the resulting representation strictly preserves the original chronological order. Subsequently, within the supervised learning context, we introduced two types of graph representations, Data Splits and Observation Layers, to handle edges that cross temporal splits (i.e. edges that share a node but originate from temporal edges belonging to different temporal windows). This approach effectively prevents data leakage while maintaining temporal consistency in the representation.

RQ2: Complexity Trade-offs. What are the implications of this approach regarding computational resources? Specifically, does the reduction in architectural complexity offered by static GNNs justify the increased computational cost associated with processing the larger, mapped input graphs?

A2: The current implementation represents a reasonable trade-off for a preliminary analysis. However, to provide a more robust and justified answer, experiments on a wider range of datasets would be necessary, as our only trial dataset Wikipedia is to be considered small (see 4.1.1).

Excluding the transformation procedure, which only needs to be performed once and is not computationally intensive, the main bottleneck arises from generating training negative edges, as these must be computed at each epoch using a relatively sophisticated technique. (see Appendix B for the implementation details and 3.2 for the theoretical definition).

Two strategies can be considered to mitigate this issue: (1) selecting datasets with fewer negative edges to generate, or (2) implementing a simplified negative edge generation procedure that requires fewer checks. This choice can be justified probabilistically, with potential negative destinations now following a strict historical paradigm or purely random selection, rather than the previously used relaxed historical paradigm, depending on the context. While this latter approach may reduce certain controls, it could introduce additional nodes and require new management strategies.

RQ3: Methodological Efficacy. Can a Supra-Adjacency representation effectively encode temporal dependencies into a static topological structure to solve supervised tasks (in our particular case a Dynamic Link Prediction), with performance comparable to native dynamic models?

A3: Based on our preliminary results, the supra-adjacency representation appears only partially effective for encoding temporal dependencies within a static structure for Dynamic Link Prediction. The limited performance is likely due to dataset constraints and the absence of informative features that could better capture temporal patterns. In our favor, prior work by the current top-ranked model for the Wikipedia dataset, Lu et al. [30][1], emphasizes the importance of exploiting the graph structure[2] even in temporal settings to achieve higher performance.

[1]Reported according to the most recent leaderboard data, November 2025.

[2]Note: They refer to the original temporal graph. However, we consider this insight relevant for our static graph representation as well.

Several improvements could enhance our approach. First, employing datasets that provide richer node and edge features, denser activity per node (ideally allowing a node to perform multiple actions at the same timestamp), and fewer nodes appearing after the learning timeline (i.e. a less inductive context) could produce more expressive static representations. Second, the graph could be enriched with temporal-aware edge features, such as encoding the time difference between the $(node, timestamps)$ of an edge as $1/\Delta t$, which introduces both temporal ordering and effective event time-proximity relevance, concepts that are crucial for dynamic tasks.

Although this model may not be optimal for Dynamic Link Prediction, it could prove to be more effective for other supervised tasks, depending on the representation and temporal information incorporated.

In conclusion, this master's thesis established a foundation for applying classical GNNs to transformed-to-static dynamic graphs. Even though this preliminary approach did not succeed in developing a definitive, fully optimized encoder to exploit the graph structure, it provides a baseline for future work in this area and achieves moderate results compared to other architectures.
The Wikipedia dataset likely represents a particularly challenging scenario, which nonetheless highlighted critical issues that might otherwise gone unnoticed. These challenges could be addressed in future studies, and this work serves as an initial step toward more effective methodologies.

Bibliography

[1] Réka Albert and Albert-László Barabási. "Statistical mechanics of complex networks". In: *Reviews of Modern Physics* 74.1 (2002), pp. 47–97. DOI: 10.1103/RevModPhys.74.47. URL: https://doi.org/10.1103/RevModPhys.74.47.

[2] Albert-László Barabási and Márton Pósfai. *Network Science*. Cambridge, UK: Cambridge University Press, 2016. ISBN: 9781107076266. URL: https://networksciencebook.com/.

[3] Yoshua Bengio and Jean-SÉbastien Senecal. "Adaptive Importance Sampling to Accelerate Training of a Neural Probabilistic Language Model". In: *IEEE Transactions on Neural Networks* 19.4 (2008), pp. 713–722. DOI: 10.1109/TNN.2007.912312.

[4] Complex Data Lab. *PyGLinkPropPredDataset.load_test_ns – Temporal Graph Benchmark Documentation*. https://docs.tgb.complexdatalab.com/api/tgb.linkproppred/#tgb.linkproppred.dataset_pyg.PyGLinkPropPredDataset.load_test_ns. Accessed: 2025-11-04.

[5] PyTorch Contributors. *torch.nn.BCEWithLogitsLoss — PyTorch Documentation*. Accessed: 2025-11-03. 2025.

[6] Nick Craswell. "Mean Reciprocal Rank (MRR)". In: *Encyclopedia of Database Systems*. Springer, Boston, MA, 2009, p. 1703. DOI: 10.1007/978-0-387-39940-9_488.

[7] Reinhard Diestel. *Graph Theory*. 6th ed. Vol. 173. Graduate Texts in Mathematics. Berlin, Heidelberg: Springer, 2025. ISBN: 978-3-662-70107-2. DOI: 10.1007/978-3-662-70107-2. URL: https://link.springer.com/book/10.1007/978-3-662-70107-2.

[8] Manuel Dileo, Matteo Zignani, and Sabrina Gaito. "Temporal graph learning for dynamic link prediction with text in online social networks". In: *Machine Learning* 113.4 (2024), pp. 2207–2226. DOI: 10.1007/s10994-023-06475-x. URL: https://doi.org/10.1007/s10994-023-06475-x.

[9] Matthias Fey and Jan Eric Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *arXiv preprint arXiv:1903.02428* (2019). arXiv:1903.02428.

[10] Edoardo Galimberti et al. "Mining (maximal) span-cores from temporal networks". In: *CoRR* abs/1808.09376 (2018). arXiv: 1808.09376 [cs.SI].

[11] Jian Gao, Jianshe Wu, and JingYi Ding. "HyperEvent: Learning Cohesive Events for Large-scale Dynamic Link Prediction". In: *arXiv preprint arXiv:2507.11836* (2025). Submitted 16 July 2025. URL: https://arxiv.org/abs/2507.11836.

[12] Justin Gilmer et al. "Neural message passing for quantum chemistry". In: *Proceedings of the 34th International Conference on Machine Learning (ICML)*. 2017, pp. 1263–1272.

[13] Aditya Grover and Jure Leskovec. "node2vec: Scalable Feature Learning for Networks". In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM. 2016, pp. 855–864. DOI: 10.1145/2939672.2939754. URL: https://cs.stanford.edu/~jure/pubs/node2vec-kdd16.pdf.

[14] Will L. Hamilton, Zhitao Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*. 2017, pp. 1024–1034.

[15] William L. Hamilton. *Graph Representation Learning*. Vol. 14. Synthesis Lectures on Artificial Intelligence and Machine Learning 3. Morgan & Claypool Publishers, 2020. DOI: 10.2200/S01045ED1V01Y202009AIM046.

[16] Petter Holme and Jari Saramäki. "Temporal networks". In: *Physics Reports* 519.3 (Oct. 2012), pp. 97–125. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2012.03.001. URL: http://dx.doi.org/10.1016/j.physrep.2012.03.001.

[17] Kurt Hornik. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366.

[18] Mohammad Mehdi Hosseinzadeh et al. "Temporal networks in biology and medicine: a survey on models, algorithms, and tools". In: *Network Modeling Analysis in Health Informatics and Bioinformatics* 12.10 (2023). DOI: 10.1007/s13721-022-00406-x.

[19] Charles Tapley Hoyt et al. "A Unified Framework for Rank-based Evaluation Metrics for Link Prediction in Knowledge Graphs". In: *Proceedings of TheWebConf Workshop on Graph Learning Benchmarks (WWW22)*. arXiv:2203.07544. 2022.

[20] Shenyang Huang et al. “Temporal Graph Benchmark for Machine Learning on Temporal Graphs”. In: *Advances in Neural Information Processing Systems (NeurIPS) 2023 Datasets and Benchmarks Track*. Datasets available at TGB website. 2023. URL: https://tgb.complexdatalab.com/.

[21] Shenyang Huang et al. *TGBL-Wiki v2: Temporal Graph Benchmark – Wikipedia co-editing dataset*. Dataset version 0.7.5, available from https://tgb.complexdatalab.com/docs/linkprop/#tgbl-wiki-v2. Temporal Graph Benchmark (TGB) link property prediction module. 2023.

[22] Shenyang Huang et al. “UTG: Towards a Unified View of Snapshot and Event Based Models for Temporal Graphs”. In: *Proceedings of the Third Learning on Graphs Conference*. Vol. 269. Proceedings of Machine Learning Research. PMLR, 2025, 28:1–28:16. URL: https://proceedings.mlr.press/v269/huang25a.html.

[23] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *International Conference on Machine Learning (ICML)*. 2015.

[24] K. Z. Khanam, G. Srivastava, and V. Mago. “The homophily principle in social network analysis: A survey”. In: *Multimedia Tools and Applications* 82 (2023), pp. 8811–8854. DOI: 10.1007/s11042-021-11857-1. URL: https://link.springer.com/article/10.1007/s11042-021-11857-1.

[25] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *International Conference on Learning Representations (ICLR)*. arXiv:1609.02907. 2017.

[26] Emma Kondrup. “Base3: a simple interpolation-based ensemble method for robust dynamic link prediction”. In: *arXiv preprint arXiv:2506.12764* (2025). DOI: 10.48550/arXiv.2506.12764.

[27] Srijan Kumar, Xikun Zhang, and Jure Leskovec. “Predicting dynamic embedding trajectory in temporal interaction networks”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM. 2019, pp. 1269–1278.

[28] Antonio Longa et al. “Graph Neural Networks for Temporal Graphs: State of the Art, Open Challenges, and Opportunities”. In: *arXiv preprint arXiv:2302.01018* (2023). Version v4, July 11 2023. URL: https://arxiv.org/abs/2302.01018.

[29] Haoran Lu et al. “A survey of graph neural networks and their industrial applications”. In: *Neurocomputing* 614 (2025), p. 128761. DOI: 10.1016/j.neucom.2024.128761. URL: https://www.sciencedirect.com/science/article/pii/S0925231224015327.

[30] Xiaodong Lu et al. “Improving Temporal Link Prediction via Temporal Walk Matrix Projection”. In: *Advances in Neural Information Processing Systems (NeurIPS) 37*. 2024. URL: https://proceedings.neurips.cc/paper_files/paper/2024/file/ff7bf6014f7826da531aa50f4538ee19-Paper-Conference.pdf.

[31] Yuhong Luo and Pan Li. “Neighborhood-aware Scalable Temporal Network Representation Learning”. In: *arXiv preprint arXiv:2209.01084v3* (2022). Last revised 30 Nov 2022.

[32] Xiaojie Ma et al. “A Comprehensive Survey on Graph Anomaly Detection with Deep Learning”. In: *arXiv preprint arXiv:2106.07178* (2021). Available at: https://arxiv.org/abs/2106.07178.

[33] Tomas Mikolov et al. “Efficient Estimation of Word Representations in Vector Space”. In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2013. URL: https://arxiv.org/abs/1301.3781.

[34] Lutz Oettershagen et al. *Temporal Graph Kernels for Classifying Dissemination Processes*. 2021. arXiv: 1911.05496 [cs.SI]. URL: https://arxiv.org/abs/1911.05496.

[35] Aldo Pareja et al. “EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs”. In: *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, 2020, pp. 5363–5370. DOI: 10.1609/aaai.v34i04.5984.

[36] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. “DeepWalk: Online Learning of Social Representations”. In: *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. ACM. 2014, pp. 701–710. DOI: 10.1145/2623330.2623732. URL: https://research.google/pubs/pub45669/.

[37] Simone Piaggesi and André Panisson. “Time-varying graph representation learning via higher-order skip-gram with negative sampling”. In: *EPJ Data Science* 11.1 (2022), p. 33. DOI: 10.1140/epjds/s13688-022-00344-8.

[38] Farimah Poursafaei et al. “Towards Better Evaluation for Dynamic Link Prediction”. In: *NeurIPS Datasets and Benchmarks Track*. OpenReview preprint, https://openreview.net/forum?id=1GVpwr2Tfdg. 2022.

[39] Emanuele Rossi et al. “Temporal Graph Networks for Deep Learning on Dynamic Graphs”. In: *arXiv preprint arXiv:2006.10637* (2020).

[40] Diego Saldaña-Ulloa, Guillermo De Ita Luna, and J. Raymundo Marcial-Romero. “A Temporal Graph Network Algorithm for Detecting Fraudulent Transactions on Online Payment Platforms”. In: *Algorithms* 17.12 (2024), p. 552. DOI: 10.3390/a17120552. URL: https://www.mdpi.com/1999-4893/17/12/552.

[41] Aravind Sankar et al. “DySAT: Deep Neural Representation Learning on Dynamic Graphs via Self-Attention Networks”. In: *Proceedings of the 13th ACM International Conference on Web Search and Data Mining (WSDM '20)*. Houston, TX, USA: ACM, 2020, pp. 519–527. DOI: 10.1145/3336191.3371845.

[42] Koya Sato et al. “Predicting partially observed processes on temporal networks by Dynamics-Aware Node Embeddings (DyANE)”. In: *EPJ Data Science* 10.1 (2021), p. 22. DOI: 10.1140/epjds/s13688-021-00277-8.

[43] Franco Scarselli et al. “The graph neural network model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80.

[44] Temporal Graph Benchmark. *TGB Dataset Overview*. https://tgb.complexdatalab.com/docs/dataset_overview/. Accessed: YYYY-MM-DD.

[45] Temporal Graph Benchmark (TGB). *Leaderboards for Dynamic Link Property Prediction — TGB*. 2025. URL: https://tgb.complexdatalab.com/docs/leader_linkprop/#tgbl-wiki-v2 (visited on 11/22/2025).

[46] Josephine M. Thomas et al. “Graph Neural Networks Designed for Different Graph Types: A Survey”. In: *Transactions on Machine Learning Research* (2023). CC BY 4.0. URL: https://openreview.net/forum?id=h4BYtZ79uy.

[47] Yifeng Tian et al. “Data-Driven Modeling of Dislocation Mobility from Atomistics Using Physics-Informed Machine Learning”. In: *arXiv preprint arXiv:2403.14015* (2024). arXiv:2403.14015v1 [cond-mat.mtrl-sci]. DOI: 10.48550/arXiv.2403.14015.

[48] *torch.nn.BatchNorm1d — PyTorch Documentation*. https://docs.pytorch.org/docs/stable/generated/torch.nn.BatchNorm1d.html. Accessed: 2025-11-03.

[49] Rakshit Trivedi et al. “DyRep: Learning Representations over Dynamic Graphs”. In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=HyePrhR5KX.

[50] Petar Veličković et al. “Graph Attention Networks”. In: *International Conference on Learning Representations (ICLR)*. arXiv:1710.10903. 2018.

[51] Yanbang Wang et al. *Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks (CAW)*. https://snap.stanford.edu/caw/. Accessed: 2025-11-24. 2021.

[52] Da Xu et al. *Inductive Representation Learning on Temporal Graphs*. 2020. arXiv: 2002.07962 [cs.LG]. URL: https://arxiv.org/abs/2002.07962.

[53] Le Yu et al. “Heterogeneous Graph Representation Learning with Relation Awareness”. In: *IEEE Transactions on Knowledge and Data Engineering* (2022), pp. 1–1. ISSN: 2326-3865. DOI: 10.1109/tkde.2022.3160208. URL: http://dx.doi.org/10.1109/TKDE.2022.3160208.

[54] Le Yu et al. “Towards Better Dynamic Graph Learning: New Architecture and Unified Library”. In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=xHNzWHbklj.

[55] Chuxu Zhang et al. “Heterogeneous Graph Neural Network”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD 2019)*. 2019, pp. 793–803. DOI: 10.1145/3292500.3330961.

[56] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. 2021. arXiv: 1812.08434 [cs.LG]. URL: https://arxiv.org/abs/1812.08434.

Appendix A

Positive edges creation

We reserve this section to illustrate the process leading to the mapping defined in Section 2.1.2, with particular focus on the specific triplet for the directed bipartite graph case described in 3.1, such as in Wikipedia.
The first relevant element in the pipeline is the creation of some variables: number of editors, wikipages and the t_{matrix}, where the latter one will be vital in our workflow and will be later adapted and utilized also in the Appendix B. The variable t_{matrix} is a vector of size total number of nodes made of sets, each one containing the lists of timestamps where a node was active: for example, given the temporal graph $G = (V, E, T)$ if we have $(i, j, t) \in E$ then

$$t \in t_{matrix}[i] \quad \text{and} \quad t \in t_{matrix}[j]. \tag{A.1}$$

where $t_{matrix}[i]$ and $t_{matrix}[j]$ are the i_{th} and j_{th} rows of t_{matrix}.

t_matrix	
Indices	Sets
$train\_node_0$	$[t_0, t_i, \dots]$
$train\_node_1$	$[t_1, t_j, \dots]$
...	...
$train\_node_N$	$[t_N, \dots]$

Figure A.1: Example of the t_{matrix} shape.

It is straightforward to observe this list facilitates the retrieval of the appropriate timestamp during the translation. It is also important to point out that in both versions of the graph mapping of Sections 3.3.1 and 3.3.2 we will need to add to each set one or more additional timestamps corresponding to

the Observation Layers timestamps. In particular we will have the following additions:

- In **Data Splits approach** (Paragraph 3.3.1) we will add to each set the timestamp $t_{obs\_lyr} = t_{max,test} + 1$ indiscriminately;

- In **Observation Layers approach** (Paragraph 3.3.2) instead we have to apply up to three additional timestamps: $t_{obs\_lyr,train}, t_{obs\_lyr,val}$ and $t_{obs\_lyr,test} = t_{obs\_lyr}$ of the Data Splits model. The presence of one or more timestamps in a set is largely determined by the effective activity of a node. If a set contains timestamps $t \in T_{train}$, we add the "closing and observational" time $t_{obs\_lyr,train}$. The same reasoning applies to the Validation and Test splits. In the end each set will contain only the essential timestamp to fully express the node's activities, saving us from future redundant cross edges.

Defined the required variables we can start our iteration over the full set of temporal edges and we define one by one the respective triplets of self and event edges. We have implied in our discussion that to speed up the process we transformed t_{matrix} into a dictionary of ordered vectors. In this structure, the keys correspond to the nodes in the lists, and the ordered times enable faster lookup of the appropriate element.
Other variables we create during the mapping phase are the split vectors, each containing a flag for each mapped edge $\big[(src, t), (dst, t')\big]$ to sign whether the built (src, t) and (dst, t') belong or not to the same time split. This will help us in the Data Splits approach where we need to identify the cross self-edges that have to be moved into the respective destination split as explained briefly in Section 3.3.1. On the other hand, since each temporal edge is mapped one-to-one to its corresponding triplet, and the resulting sequences of edges are stored in order for easier manipulation, we must remember to explicitly define the cross-self edges when constructing the Observation Layers graph representation (see Paragraph 3.3.2). This process is done verifying first the presence of $t_{obs\_lyrs,i}$, (with i = Train, Validation, Test) and then generating the consequent cross self edges.
After completing the full mapping process, we can define the sets of (src, t) and (dst, t) nodes. These collections form the basis for the dictionaries of unique IDs required by our architecture to process the mapped graph, as described in Section 3.4.

Finally, it is worth noting that, although we could already remove repeated self-edges at this stage, we deliberately postpone this operation to the analysis phase. In the Wikipedia dataset case their number is limited (i.e. not

computationally costly) and they may carry additional information for alternative approaches not considered in this work.

Appendix B

Negative edges creation

The original TGB Dataset documentation [4] is equipped with a series of function that automatize the creation of temporal negative edges for the training phase while for validation and test provides list of lists, each containing all the available negative destinations for each positive temporal edge. In particular the new implementation of Wikipedia has defined around 999 negative destinations per positive edge (see Paragraph 3.2), the highest possible number to enforce a more demanding task.
For our discussion we adapted the given validation and test negative edges to our translation process, while for the training negative edges we constructed from scratch a set of functions and auxiliary variables to achieve our goal, accounting for the possible variants allowed by our static graph version.

B.1 Validation and Test negative edges

First, we describe the validation and test translation into the static graph version due to their simpler mapping. As previously mentioned we can load through the TGB's functions the whole sets of negative destinations each original positive edge (i, j, t) the graph has. Once we download the whole list we can derive the full list of (src, dst_{neg}, t) that we can feed in the same system we used for the positive edges (see Appendix A) with the only detail that only the event edge mapping is required. It is important to note that, based on the definition provided in Section 3.2, the entire set of these edges is considered to be made of only Hard negative edges. This choice ensures both easier translation and closer alignment with the positive edges, resulting in a more effective source of information during the validation phase.
We finally conclude our translation mapping each tuple $(node, t)$ to its proper unique ID and building the mapped list of negative destinations back into

their list of lists format.

B.2 Train negative edges

The work necessary to properly define the train negative edges is comparable to or exceeding the mapping of the positive edges mapping (see Appendix A and Eq. 2.10 as reference): in this case we reused some concepts used in the positive mapping and we readapt them for this other conversion.
Likewise the positive mapping, our process starts defining a variable t_{matrix} (see Fig. A.1), a vector of size equal to the number of all the possible destinations (in the Wikipedia dataset case 1,000), where its i_{th} element contains the set of timestamps in which the i_{th} destination has been active: if $(i, j, t) \in E$ then $t \in t_{matrix}[j]$ (see Eq. A.1 notation). We repeat ourselves in saying this variable is vital to re-use only the already existing nodes: if not, we would fall in a case where every possible $(node, timestamp)$ tuples is available, leading to dramatic consequences for the size we work on and the variability of choice.
The following auxiliar variables we generate are rem (removal) and rem_{tuples}, with an analogous format of t_{matrix} but of size positive event edges. In this case for each positive edge (to be more precise each (src, t) coming from the positive event edge map source) we save the list of destinations dst and tuples (dst, t'), respectively for rem and rem_{tuples}, coinciding with the all available $[(src, t), (dst, t')]$ defined as event edges during the positive mapping. This parameter will help us to quickly identify whether a potential negative edge is erroneously a positive one and discard it. A second note is that in Wikipedia most sets contain only a single element due to the near uniqueness of events occurring at each timestamp (see Table 4.1). Nevertheless, we retain this structure for the sake of generality.

rem	
Indices	Sets
(src_0, t_0)	$[dst_0, dst_x]$
(src_1, t_1)	$[dst_1]$
...	...
(src_N, t_N)	$[dst_N]$

(a) rem variable representation

rem_tuples	
Indices	Sets
(src_0, t_0)	$[(dst_0, t_0), (dst_x, t_x)]$
(src_1, t_1)	$[(dst_1, t_1)]$
...	...
(src_N, t_N)	$[(dst_N, t_N)]$

(b) rem_{tuples} representation

Figure B.1: rem and rem_{tuples} format representation.

Reached this point we define two distinctive functions, one to build Hard negatives and one to build simultaneously Weak and Very Weak negative edges

B.2.1 Hard negatives

As previously explained in Paragraph 3.2 and the previous Appendix B.1, Hard train negative edges are easily formed as if they were event edges in the positive case (see Appendix A): taken a positive temporal edge (src, dst, t) and consequent event edge $[(src, t), (dst, t')]$ we choose randomly a list of potential dst_{neg} we immediately check with rem to discard any potential original temporal edge. Once this first check is passed, we exploit t_{matrix} to locate, in the set corresponding to dst_{neg}, the first timestamp $t_{neg} > t$. We then use this value to construct (dst_{neg}, t_{neg}), which maps the negative destination according to the original rules in Eq. 3.1. As repeatedly stated this tuple finally will be first mapped into its unique IDs and then placed into the translated set of negatives.
It is important to note that (to avoid any kind of errors or creation problem) it is required that each set of t_{matrix} contains at least one element bigger than the maximum value of the $T_{train} = \{t \in (i, j, t) \in T_{train}\}$ (in Wikipedia this is reached only exploiting the timestamps coming from the event edges of the train split).

Another detail specific to Wikipedia is that the number of training negative destinations does not coincide with the total number of destinations (i.e. 952 training nodes out of 1,000 in total). This occurs because not all nodes are involved in the training phase (see Fig. 4.1), and we must ensure that nodes unseen during training are not used as potential negative destinations. For this reason, the selection of potential dst_{neg} should not be described as "random," but rather as a "relaxed historical" selection. Specifically, we consider only nodes that appeared during the training phase. However, unlike the original "pure historical" definition [38], these nodes are not required to have edges with the selected *source* node within this temporal window, which gives rise to their name (see Paragraph 3.2).

B.2.2 Weak negatives

This final function has been created specifically to build the missing negative edges: as shown in Paragraph 3.2, our translation give us the opportunity to define a great variety of negatives respect the original counterpart. It is important also to add this is a fundamental piece to properly create the sufficient number of negatives (in Wikipedia is 999), otherwise impossible by

what the restricted 952 destinations training allow us and that we cannot increase artificially without the risk of incurring in data leakage.
With this in mind, we proceed first defining the variables that help us into building the right ratios of Weak or Very Weak negatives, then collecting the lists of potential negative destinations. It is important to emphasize that in this case, the control to discard unintended edges is performed only at the end of the process, since any available destination may still form a non-existent edge (i.e. different from the positive one) until the final step, unlike the Hard case described in Appendix B.2.1. Then, depending on the variant to be generated, we sample a random t_{neg} for each dst_{neg}. In the Weak case, it is selected at random under the constraint $t_{neg} > t$; in the Very Weak case, t_{neg} is selected uniformly from the available timestamps in t_{matrix}; The process ends with the not positivity check, the translation into the corresponding ID and the addition to the count of one element, keeping control of the required ratios.

Appendix C

Architectures

After describing all the necessary steps to properly map a temporal graph, we can present the architectures chosen to address the Link Prediction task. We describe the most general implementations of the encoder, normalization, decoder, and evaluation metric here, as they are applied in both the Data Splits and Observation Layers approaches, with some variants discussed in the previous chapters.

C.1 Encoder

The chosen Graph Neural Network (GNN) is going to be an **HeteroSAGE** model [55, 47]. In particular one layer of this architecture performs four message-passing, one for each kind of edge: source self-edges (Wikipedia editor), destination self-edges (Wikipedia wikipages), event edges and reversed event 3.4 (due to the required Link Prediction, Section 3.2).
As a standard approach, the convolutions over different edge types are handled by HeteroConv, a wrapper module provided by PyTorch Geometric [9]. Finally, as the name itself suggests, we opted to perform message aggregation using GraphSAGE/SAGEConv (see Section 2), a simple well known model for being inductive and scalable [14].
More formally we can express the illustrated layer in the following way: Given the set of relations $R = \{self_{src}, self_{dst}, event, rev\_event\}$ (as formalized in Paragraph 3.4) and defining $j \in \mathcal{N}_i^{(r)}$ as the neighbours of node i connected with it by an edge of relation $r \in R$, we can define the aggregation step of the SAGEConv in various ways. In our particular work we focus on the AGGREGATE function

$$\mathbf{h}_{\mathcal{N}_i^{(r)}} = \frac{1}{|\mathcal{N}_i^{(\mathrm{r})}|} \sum_{j \in \mathcal{N}_i^{(r)}} \mathbf{x}_j^{(0)}, \tag{C.1}$$

which is the default MEAN of the neighbor features and the node i's self features (in particular here $x_j^{(0)}$ are the initial node embeddings).

As mentioned, our HeteroSAGE consists of four types of SAGEConv layers, one for each relation, which are combined into a single HeteroConv layer. This layer simply **sum** the contributions from all relation types $r \in R$ for each node i (in our code, this is specified by aggr='sum' [1]):

$$\mathbf{h}_i^{(t)} = \sum_{r \in \mathcal{R}_i} \sigma\left(\mathbf{W}_{\text{self}}^{(r)} \mathbf{x}_i^{(t-1)} + \mathbf{W}_{\text{neigh}}^{(r)} \cdot \mathbf{h}_{\mathcal{N}_i^{(r)}}\right) \tag{C.2}$$

where:

- $\mathcal{R}_i$ is the set of all relations r where i is the destination node;
- $\mathbf{x_i}^{(t-1)}$ and $\mathbf{x_j}^{(t-1)}$ are the node and neighbor embeddings from the previous layer;
- $\mathbf{W}_{\text{self}}^{(r)}$ and $\mathbf{W}_{\text{neigh}}^{(r)}$ are the learnable weight matrices specific to relation r for the node and its neighbours;
- $\sigma(\cdot)$ is the non-linear activation function (e.g. ReLU, applied after normalization in your model's forward pass).

C.2 Normalization

In between the HeteroConv layers we apply **self.norm1** layers, known as **Per-Node-Type Batch**, a batch Normalization taken from PyTorch ModuleDict [48]. Batch Normalization (BatchNorm1d) for a feature vector $\mathbf{x}$ within a mini-batch $\mathcal{B}$ is defined by normalizing $\mathbf{x}$ based on the batch statistics [23]:

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \tag{C.3}$$

This normalized vector is then scaled and shifted by learnable parameters γ and β:

$$\mathbf{y} = \gamma \hat{\mathbf{x}} + \beta \tag{C.4}$$

[1] We did not perform a definitive evaluation, but across various approaches this appeared to be the best option. Further confirmation is requested and encouraged.

where $\mu_{\mathcal{B}}$ and $\sigma^2_{\mathcal{B}}$ are the mean and variance of the feature dimension across the current batch $\mathcal{B}$, and ϵ is a small constant for numerical stability.
Its presence is necessary to stabilize and accelerate training while handling feature differences across the heterogeneous node types like our case.

C.3 Decoder

The other piece common in all our architectures is the decoder, a **Multi-Layer Perceptron** (**MLP**) [17].
The goal of the MLP is to compute a score $S(\mathbf{z}_{src}, \mathbf{z}_{dst})$ which indicates the probability of the existence of a link between a source node having embeddings z_{src} and a destination one with embeddings z_{dst}, where z_{src}, $z_{dst} \in \mathbb{R}^d$ ($d - dim$ Euclidean space, with $d =$ hidden channels size). It is obvious this is the required Link Prediction task (see Section 3.2).

In order to evaluate the score $S(\mathbf{z}_{\text{src}}, \mathbf{z}_{\text{dst}})$ it is necessary to first create the enriched input vector $\mathbf{x}$ and our choice has lead to the more expressive and simple four elements concatenation:

$$\mathbf{x} = \text{Concat}(\mathbf{z}_{\text{src}}, \mathbf{z}_{\text{dst}}, \mathbf{z}_{\text{sum}}, \mathbf{z}_{\text{hadamard}}) \in \mathbb{R}^{4d} \tag{C.5}$$

where we recognize the source and destination embeddings z_{src}, z_{dst} and the two common link prediction operators: the element-wise sum and the element-wise Hadamard (product), defined respectively as

- **Element-wise Sum:** $\mathbf{z}_{\text{sum}} = \mathbf{z}_{\text{src}} + \mathbf{z}_{\text{dst}} \in \mathbb{R}^d$
- **Hadamard Product:** $\mathbf{z}_{\text{hadamard}} = \mathbf{z}_{\text{src}} \odot \mathbf{z}_{\text{dst}} \in \mathbb{R}^d$

Computed the enriched input vector $\mathbf{x}$, it can be fed into our **MLP**, composed by two layers, where the first one uses both a Rectified Linear Unit (ReLU) and Dropout:

$$S(\mathbf{z}_{src}, \mathbf{z}_{dst}) = \mathbf{W}_2^\top \cdot \text{Dropout}(\text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)) + b_2 \tag{C.6}$$

where we identify

- $\mathbf{W}_1 \in \mathbb{R}^{h \times 4d}$ and $\mathbf{b}_1 \in \mathbb{R}^h$ are the weights and non-linear term of the first linear layer;
- $\mathbf{W}_2 \in \mathbb{R}^{h \times 1}$ and $b_2 \in \mathbb{R}$ are the weights and non-linear term of the second and final linear layer;

- d = Node embeddings size;
- h = Hidden layer size.

C.4 Loss Function

We introduce one of the remaining components necessary to complete our link prediction framework: the loss function. Given the strong class imbalance between negative and positive samples (approximately 999:1 negative-to-positive ratio in the Wikipedia dataset), we adopt a **Weighted Binary Cross-Entropy with Logits Loss** to compensate for this imbalance and stabilize training.

By definition, the loss function $\mathcal{L}$ is computed as the average of the individual loss terms $\mathcal{L}_i$ across the mini-batch [5]:

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i(s_i, y_i, W) \tag{C.7}$$

where the individual loss term $\mathcal{L}_i$ for the i-th link prediction sample is defined as:

$$\mathcal{L}_i(s_i, y_i, W) = -\left[y_i \cdot W \cdot \log(\sigma(s_i)) + (1 - y_i) \cdot \log(1 - \sigma(s_i))\right] \tag{C.8}$$

Specifically:

- **N**: total number of links (both positive and negative) in the current mini-batch;
- $\mathbf{s_i}$: raw score (logit) output by the model for the i-th link, obtained using the MLP decoder (see Section C.3);
- $\mathbf{y_i}$: ground-truth label for the i-th link, where $y_i \in \{0, 1\}$;
- $\sigma(\mathbf{s_i})$: sigmoid activation function, $\sigma(s_i) = \frac{1}{1+e^{-s_i}}$, which converts the raw score s_i into a predicted probability;
- **W**: positive-class weight used to address class imbalance. It scales the loss contribution of the positive class ($y_i = 1$). In our case, $W = \frac{\text{Number of Negative Samples}}{\text{Number of Positive Samples}}$.

C.5 Evaluation Metric

The TGB datasets involving Dynamic Link Prediction such as Wikipedia dataset use as the standard evaluation metric the **Mean Reciprocal Rank (MRR)**. This metric measures how highly a correct (positive) link is ranked among a set of candidate negative links [6, 19]. As the name suggests **MRR** is a mean of Reciprocal ranks **(RR)**, a parameter defined as the inverse of the rank position of the first true positive element:

$$RR_i = \frac{1}{\text{rank}_i} \tag{C.9}$$

It is easy to see $RR_i \in (0, 1]$, where the highest value (i.e 1) can be reached only imposing the right element in the first place.

Because evaluation is performed over mini-batches of queries, we compute the reciprocal rank RR_i of each query i i independently, and subsequently average these values to obtain the batch-level performance:

$$MRR = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{\text{rank}_i} \tag{C.10}$$

Similarly to what said for the singular RR_i the best result will be obtained to a score close or equal to 1, symbol of the fact our model can correctly recognize and rank into the first places the correct edge (that in our task has to be predicted).