UNIVERSITY OF TURIN

Master Degree Course in Physics of Complex Systems

Master Degree Thesis

Graph Representation Learning applied to Financial Transaction Networks

Supervisors
André Panisson
Michele Starnini

Candidate
Claudio Moroni

Academic Year 2022-2023

Summary

Financial Transaction Networks are directed, temporal heterogeneous graphs representing payments between bank accounts. They are among the most important sources of data for banks to study customer behavior and deploy efficient, neural Anti-Money Laundering systems. Moreover, being transaction networks a very scarce type of data (due to privacy and business interests), Synthetic Data Generation is also worth exploring. Nonetheless, the study of representation learning techniques for directed temporal networks is an active research area. Therefore, we aim to give the following contributions: a self-contained explanation of related background concepts otherwise very sparsely presented in the literature, and contribute a few technical steps towards the development of a representation learning framework for directed temporal graphs. Regarding background concepts, we will first review deep learning techniques over static and dynamic graphs, and we will extensively expand on concepts such as Variational Graph Autoencoders and Model Agnostic Meta Learning. Next, we will discuss the issues related to transaction network's directionality and temporality separately, by proposing two representation learning frameworks. Regarding directionality, we will tackle Directed Link Prediction as an early approach to deep directed graph generation. An acceptable model for Directed Link Prediction should not only predict the existence of an edge between two nodes, but also its direction and potential bidirectionality. Most recently proposed models for Directed Link Prediction are only tested in scenarios where most (if not all) negative edges are not the reverses of the positives, and the fraction of bidirectional edges is very low. In such cases, we prove that even a model structurally incapable of predicting the edges' directions nor classifying them as bidirectional or unidirectional would actually perform deceivingly well. The very few previous papers that recognize this issue fail to produce models capable of simultaneously predicting an existing edge's direction, classify it as bidirectional or not, and predicting its unconditional existence. We argue

that this happens when the four combinations of positive/negative unidirectional/bidirectional edges do not contribute equally to the training loss. Therefore, we propose a novel training technique that, by mapping Directed Link Prediction to a four-classes classification task, allows models to better embed (bi-)directionality while still retaining (and even improving) general-purpose directed link prediction capabilities in the settings models are usually tested against. The framework applies to all directed link prediction models. We show our framework's ability to properly address Directed Link Prediction in dedicated experiments. Regarding temporality, current techniques for dynamic graph representation learning aim to develop composite architectures that deal with temporal graphs directly, known as Temporal Graph Neural Networks. Inspired by a few early works, we explored the inverse approach, where static representations of dynamic graphs that preserve the temporal information are computed, to be later embedded using a Graph Neural Network. While this idea is not completely new, it has been vastly overlooked and, to the best of our knowledge, this is the first work that proposes to systematically extend this framework to all dynamic graph-related tasks. For the first time, we test this approach in a temporal node classification task and demonstrate its superiority with respect to Temporal Graph Neural Networks.

We argue that the novel training technique for Directed Link Prediction together with the Static Representations-based framework for dynamic networks constitute necessary technical steps in the direction of a Temporal Graph Representation Learning model for Transaction Networks, to address, for example, the aforementioned Anti-Money Laundering and Synthetic Data Generation needs.

Contents

Chapter 1

Introduction

Networks are most often recognized as a natural means to represent physical systems [40]. There are two types of systems that are conveniently modelled by networks: *natural networks* and *representational networks*. Natural networks are systems manifestly resembling a graph, and they include social networks, financial transaction networks, power grids, neuronal connectivity, communication networks, abstract syntax trees [1], [38], [61]. Representational networks are systems that admit a (possibly non-intuitive) reduction to a graph, such as time series [68], traffic [57] and physical systems [66]. Different systems require different types of graph representations and tasks over them. Examples of graph types are simple graphs, Knowledge Graphs [49], Dynamic Graphs [31] and multilayer graphs [10]. Instances of industrial and public issues that involve networks as underlying data structure could be predicting seismographic time series [54], designing recommender systems [69], generating new molecules [24], predicting their properties [15], detecting financial crime [50] and generating synthetic data [64]. All these issues, within the context of networks, can be naturally mapped to tasks such as node classification ([17]), node regression [54], link prediction ([65]), graph classification ([42]), subgraph matching ([37]), graph generation [72], etc.

In many of these cases, the usual way to represent a graph i.e. via its adjacency matrix or edge list is not suitable for the problem at hand, since the information needed may not be manifestly available and/or the way it has to be interpreted is not intuitive. Therefore, it is crucial to perform *Representation Learning* on graph-like data structures and also devise methods to extract predictions from the learned embeddings. The models of choice will

be Graph Neural Networks, which are the state of the art deep architectures for graphs. These models allow for end-to-end, inductive learning of nodes' representations and will be extensively studied in the remainder.

Our ultimate focus is *Transaction Networks* (*TN*s). These are attributed, timestamped, directed heterogeneous networks where nodes are bank accounts (which may vary in type i.e. private accounts, company accounts, etc) and two nodes are linked by a timestamped directed edge if they performed a transaction at a specific time. Transactions come in different forms too e.g. wire transactions, credit cards, etc., could be weighted by their amount and carry other metadata. See Figure 1.1 for an example transaction network.

The issues over TNs that concern us are *Anti-Money Laundering* and *Synthetic Data Generation*. Anti-Money Laundering consists in identifying criminal accounts involved in laundering activities (see UN Overview), while (Synthetic) Transaction Network generation is motivated by the scarcity of publicly available datasets [64]. Both tasks require developing representation learning models for TNs i.e. attributed directed temporal graphs.

Regarding Transaction Network Generation, a graph theoretic analysis of transaction networks may be found in [60]. This work highlights the need for the number of publicly available transaction network datasets to grow. Since, due to privacy and business interest, this is unlikely to happen, an alternative route would be generating them. This entails developing temporal graph generative models. We are aware of only one deep model for temporal graph generation [56], therefore we started by exploring static graph generation to identify literature gaps. Although mechanistic models for static graphs already exist (Erdős–Rényi, Watts-Strogatz, Configuration Models, Kronecker graphs), they hardly address the need to simultaneously generate both the network structure and the node/edge attributes in a statistically credible way, let alone contemplate temporality. Therefore, deep generative models are needed. Although graph diffusion models have just been developed [72], we focused on exploring Variational Graph Autoencoders (VGAEs), since they constitute a much simpler class of generative models that, as we shall prove, have not yet been fully explored in a directed setting. As usual for autoencoders, also the graph variants are made of an encoder and a decoder. The encoder is usually a Graph Neural Network that computes an embedding for each node, while the decoder can also be nonparametric and represents the law that, given the feature vectors of two nodes, computes the probability of a link between them. The associated task of correctly retrieving existing

links of a graph is known as "Link Prediction". Since Variational Graph Autoencoders introduce an element of stochasticity at prediction time, they can be employed to generate graphs by asking a trained model to predict the probability of every possible edge in a graph. Therefore, the generation of graphs' topologies can be reduced to a Link Prediction task. Just as mechanistic models, Variational Graph Autoencoders are not capable of simultaneously generating the topology together with node/edge attributes, being limited to the first. Nevertheless, they constitute a step forward since they don't make any assumption on the underlying generative process and they take node and edge features into account during training. While exploring Link Prediction in a directed setting, we noticed that related works usually don't test their models against their ability to distinguish edge direction (nor detect bidirectionality), rather, they are usually tested in settings where, despite the presence of directed edges, the statistics of their assortment makes it so that even undirected models perform decently. We describe these tasks as concerning the edges' *existence* (irrespectively of their direction). In the rare works where test sets for (bi)directionality prediction are developed, the same papers do not develop a single model capable of tackling the existence, directionality and bidirectionality tasks simultaneously. For what we said earlier, the inability to meaningfully perform Directed Link Prediction impedes the generation of directed graphs' topologies. We argue and show that mapping the directed link prediction task to a four-class edge classification task, where the classes are positive/negative unidirectional/bidirectional edges, allows training models that better compromise performance among the three sub-tasks. We name this framework, which is applicable to all directed link prediction decoders, Multiclass Framework for Directed Link Prediction (*MFDLP*). We argue that this is a necessary step towards a proper embedding of directionality information and thus towards the generation of directed networks.

Regarding Anti-Money Laundering, we know that criminals use complex transaction patterns to hide the provenance of illicit funds ([64], [27], [32]). This practice is called *Money Laundering*, and the techniques aimed at discovering bad actors are therefore collectively named *Anti Money Laundering* (*AML*). Laundering patterns are various and always evolving, as shown in Figure 1.2. It is therefore crucial to develop algorithms capable of identifying such laundering temporal patterns within the wider transaction networks. The techniques that look for pre-selected motifs with a network are known as *Subgraph Matching* techniques. A notable early work is [50], where a

statistical algorithm is developed to look for instances of two such patterns (gather-scatter and scatter-gather). Despite its success, this approach is not capable of inferring laundering patterns from the data nor leveraging the computational advantage an approximate algorithm may benefit from. For these reasons, we again believe that developing neural approaches is a necessary next step. To this end, some works on Neural Subgraph Matching over static graphs already exist ([43], [37]), but the literature about neural temporal subgraph matching is much scarcer. the natural architectures to first consider would therefore be Temporal Graph Neural Networks i.e. deep models capable of ingesting temporal graphs [70]. Anyway, TGNNs are not as principled as their static equivalents, and attaining good performance out of them is not always easy (see e.g. results on TGB). Therefore, taking inspiration from a small body of existing literature, we started tackling the problem from the opposite point of view: instead of designing architectures capable of ingesting temporal graphs, we map the temporal graph to a "static representation" such that the temporal information is encoded in the topology of the resulting network itself. Subsequently, a static Graph Neural Network is applied and its predictions are mapped back to the temporal domain. Although this idea has been pioneered in relatively recent works on temporal graph classification [41], [42], this is the first attempt to derive a temporal graph representation learning framework out of it by mapping another task (node regression/classification) and paving the way to more tasks being mapped. If successful, the advantages of this approach would be a dramatic simplification in model complexity and the establishment of a standardized approach instead of myriads of ad-hoc architectures. We envision that also the temporal subgraph matching task could be mapped, therefore providing another tool to authorities and institutions fighting Money Laundering.

We believe that the simultaneous development of the two lines of research and their eventual merger will, in the best scenario, provide a valid Representation Learning paradigm for directed temporal graphs, or at least teach much-needed lessons on the matter and provide more informed future directions.

The remainder of the thesis is organized as follows: Chapter 2 is an introductory chapter that explains the techniques behind and motivates subsequent contributions. It is comprised of the following sections: Section 2.1 introduces state-of-the-art deep models for graphs, namely *Graph Neural Networks* (*GNNs*); Section 2.2 details (Variational) Graph Autoencoders; Section 2.3 introduces techniques to scale GNNs up to large graphs; Section 2.4 illustrates Node Classification and Link Prediction tasks over graphs performed

using GNNs; Section 2.5 contains experiments and reproductions of earlier results, together with some software contributions to the ecosystem of Pytorch Geometric; Chapter 3 illustrates *MFDLP*, the proposed method to deal with directionality; Chapter 4 deals with temporal graph representation learning. It is divided in three parts: Section 4.1 follows the taxonomy of TGNNs explained in [70], and details one model per type; Section 4.2 defines common tasks over temporal graphs; Section 4.3 finally proposes our approach based on static representations to take temporality into account. Contributions come in compilatory, implementational and fully original fashions.

Compilatory contributions may be found in:

- Section 2.2.3 provides an accessible explanation of Variational Autoencoders, which attempts to clarify a rather sparse statistical literature;
- Section 4.1.2 provides the mathematical steps behind [14].

Purely implementational contributions may be found in:

- Section 2.4.2 provides a Pytorch Geometric-based implementation of [30], previously available only in older versions of Tensorflow.

Fully original contributions may be found in:

- Chapter 3 illustrates *MFDLP*;
- Section 4.3 proposes our approach based on static representations to take temporality into account.

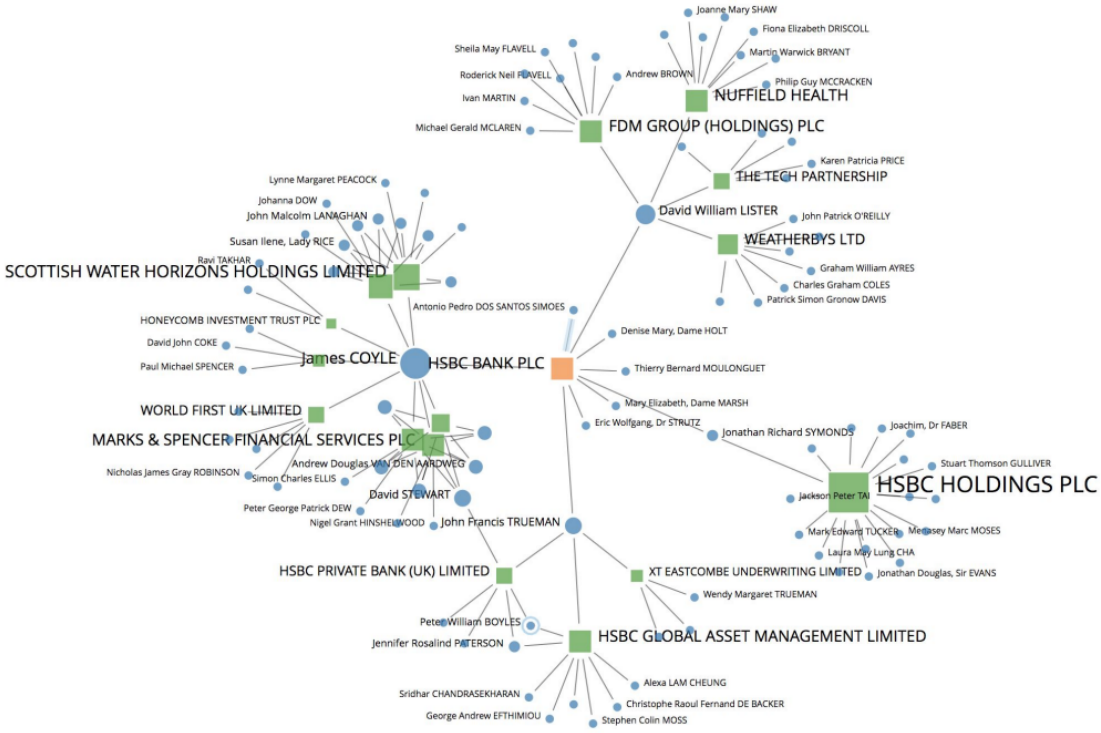


Figure 1.1: A transaction network. *Courtesy of FNA*.

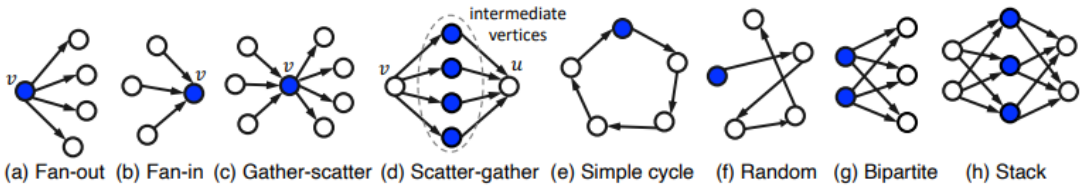


Figure 1.2: Some common laundering patterns. *Courtesy of* [64]

Chapter 2

Background

2.1 Graph Deep Learning

Whatever the modality, it is always useful to explore compatible non-deep learning architectures to develop intuition and provide useful (and oftentimes very strong) baselines. The resulting family of graph-related approaches, collectively named Graph Machine Learning (models), all exhibit the same characteristics: a transductive, feature-insensitive encoder followed by a general-purpose classifier or regressor. Due to the complexity of the tasks in this thesis, these models are largely unrelated to our purposes. Nevertheless, for the reasons expressed above, we described Graph Machine Learning models in Appendix A.

As stated in the introduction, the state-of-the-art deep learning models for graphs are known as Graph Neural Networks. The idea behind these models is to leverage the (assumed) homophily of real networks to build an end-to-end, inductive deep architecture capable of learning task-specific features of nodes. We will describe two classes of Graph Neural Networks i.e. Message Passing Neural Networks (Section 2.1.1) and Spectral Graph Convolutional Networks (Section 2.1.2) and finally argue their broad analogy.

Graph Neural Network layers (many examples of which will follow) will form the building blocks of the encoder part of the (Variational) Graph Autoencoders we will extensively use in deriving our results about Directed Link Prediction. Heterogeneous Graph Neural Networks will be employed to take

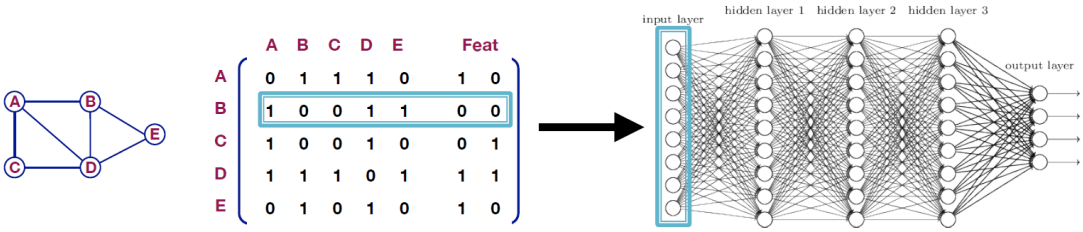


Figure 2.1: Feeding an adjacency matrix to an MLP requires choosing an arbitrary node ordering. *Courtesy of Stanford CS224W*

structural edges into account when describing Static Representations in Section 4.3.

2.1.1 Message Passing Neural Networks

The main challenge in designing neural networks for graphs resides in adapting previous architectures such as MLPs and CNNs to this new data format. A naive idea could be that of appending the node feature matrix X to the right of the adjacency matrix A, and pass the rows to an MLP.

The output could be used to train the MLP against some task.

But this approach has two problems:

1. It is sensitive to node ordering, so the trained MLP would not be applicable to a different (but equivalent!) representation of the same graph (see Figure 2.1);
2. It would not be applicable to graphs of different size.

To leverage homophily, one may think of using some CNN-like filter to aggregate node features, but a visual approach would not clarify how to slide the filter across the graph (see Figure 2.2).

Instead, it becomes clearer on how to extend CNNs to graphs if we interpret an image as a grid (a particular type of graph), where the pixels are the nodes and there is a link between two pixels if they are contiguous. The pooling operation of an $n \times m$ filter can be interpreted as computing a new "embedding" for each pixel that has at least a $n \times m$ neighbors around it (see Figure 2.3).

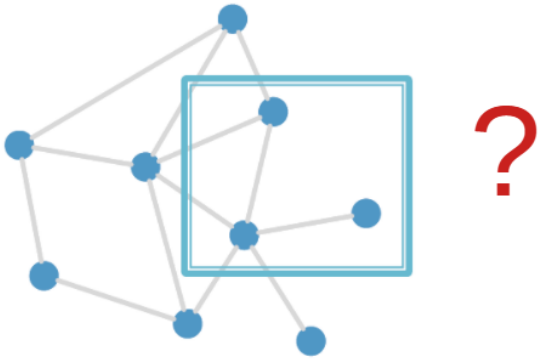


Figure 2.2: A visual approach does not help defining filters on graphs. *Courtesy of Stanford CS224W*

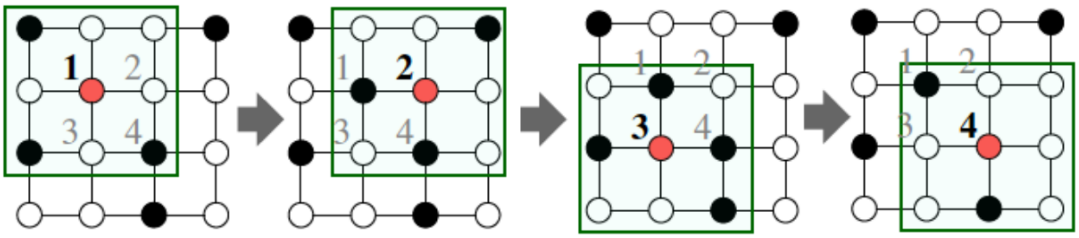


Figure 2.3: Considering pixels as nodes, and adjacency as proximity allows defining filters on networks. *Courtesy of Stanford CS224W*

Therefore, we envision a deep architecture over graphs that computes a new embedding $\vec{h}_v$ for each node v by transforming and aggregating its neighbors' $\{u \mid u \in N(v)\}$ features via learnable functions. It will be composed of K layers, the k-th of which operates on the nodes' embeddings $H^{(k-1)}$ outputted by the previous layer to produce new ones that take into account information from k-th order neighbors. We will call such architecture *Message Passing Neural Network* (*MPNN*). Concretely, suppose we are given a graph $G = (V, E)$ with nodes' feature matrix X, then the procedure by which a general MPNN of depth K would compute node v's embedding $\vec{h}_v^{(K)}$ is as follows (adapted from [15]).

Consider the neighborhood of $N(v)$ of v (different MPNNs and/or graph types might define "neighborhood" differently):

$$\{u|u \in N(v)\}$$

Apply a learnable function $\overrightarrow{\text{MSG}}^{(1)}$ to each element of the neighborhood. $\overrightarrow{\text{MSG}}^{(1)}$ will take as input the neighbor u's feature vector $\vec{h}_u^{(0)} = \vec{x}_u$, v's feature vector $\vec{h}_v^{(0)}$ and the feature vector of the edge between them $\vec{\epsilon}_{uv}$. It will output the *message* $\vec{m}_{uv}^{(1)}$ from u to v:

$$\vec{m}_{uv}^{(1)} = \overrightarrow{\text{MSG}}_{u \to v}^{(1)} = \overrightarrow{\text{MSG}}^{(1)}(\vec{h}_v^{(0)}, \vec{h}_u^{(0)}, \vec{\epsilon}_{uv})$$

Next, aggregate all messages bound for v using a permutation-invariant $\overrightarrow{\text{AGGREGATE}}^{(1)}$ function. The permutation invariance is necessary since neighbors have no fixed order. $\overrightarrow{\text{AGGREGATE}}^{(1)}$ will output the *aggregated message* $\vec{m}_v^1$ for v:

$$\vec{m}_v^{(1)} = \overrightarrow{\text{AGGREGATE}}_v^{(1)} = \overrightarrow{\text{AGGREGATE}}^{(1)}(\{\vec{m}_{uv}^{(1)}|u \in N(v)\})$$

Notice that the object $\{\vec{m}_{uv}^{(1)}|u \in N(v)\}$ is named the *message multiset*. Finally, combine the aggregated message for v again with its feature vector $\vec{h}_v^{(0)}$ using another learnable function $\overrightarrow{\text{COMBINE}}^{(1)}$ to output the first-order embedding $\vec{h}_v^{(1)}$ of v:

$$\vec{h}_v^{(1)} = \overrightarrow{\text{COMB}}_1(\vec{h}_v^{(0)}, \vec{m}_v^{(1)})$$

Algorithm 1 Message Passing Neural Network

Input: Graph $G = (V, E)$, node feature matrix $X = H^{(0)}$, edge features $\mathcal{E} = \{\vec{\epsilon}_{uv} | (u, v) \in E\}$, depth K, nonlinearities $\overrightarrow{\text{MSG}}^{(k)}, \overrightarrow{\text{COMB}}^{(k)}$, permutation-invariant function $\overrightarrow{AGGREGATE}^{(k)} \forall k = 1, ..., K$
Output: Embedding $\vec{h}_v^{(K)} \forall v \in V$
for $k = 1, ..., K$ **do**
  for $v \in V$ **do**
    for $u \in N(v)$ **do**
      $\vec{m}_{uv}^{(k)} = \overrightarrow{\text{MSG}}^{(k)}(\vec{h}_v^{(k-1)}, \vec{h}_u^{(k-1)}, \vec{\epsilon}_{uv})$
    end for
    $\vec{m}_v^{(k)} = \overrightarrow{AGGREGATE}^{(k)}(\{\vec{m}_{uv}^{(k)} | u \in N(v)\})$
    $\vec{h}_v^{(k)} = \overrightarrow{\text{COMB}}^{(k)}(\vec{h}_v^{(k-1)}, \vec{m}_v^{(k)})$
  end for
end for

The first argument of $\overrightarrow{\text{COMBINE}}^{(1)}$ will be referred to as *self-information*. This cycle is repeated until embeddings of the desired order K are computed for each node. Putting everything together in algorithmic format, we get:

We are then in the position to properly define an MPNN:

Definition 2.1.1 *A K-Message Passing Neural Network (K-MPNN) $\mathcal{M}^K$ is a K-tuple $\mathcal{M}^K = (\mathcal{M}^{(k)})_{k=1}^K = \left(\overrightarrow{MSG}^{(k)}, \overrightarrow{AGGREGATE}^{(k)}, \overrightarrow{COMBINE}^{(k)}\right)_{k=1}^K$. We will denote its action over a graph $G = (V, E)$ with node feature matrix $X = H^{(0)}$, edge features $\mathcal{E} = \{\vec{\epsilon}_{uv} | (u, v) \in E\}$ and (possibly weighted) adjacency matrix A as:*

$$H^{(K)} = \mathcal{M}^K(A, X, \mathcal{E}) \tag{2.1}$$

We may later omit some of the arguments in the right-hand side of Equation (2.1) for brevity.

Definition 2.1.2 *$\overrightarrow{MSG}^{(k)}$, $\overrightarrow{AGGREGATE}^{(k)}$ and $\overrightarrow{COMBINE}^{(k)}$ are collectively referred to as the (k-th layer's) aggregation strategy.*

After computing an embedding $\vec{h}_v^{(K)} \forall v \in V$, a permutation-invariant $\overrightarrow{\text{READOUT}}$ method can be used to return an embedding for the whole graph:

$$\vec{h}_G = \overrightarrow{\text{READOUT}}(\{\!\{\vec{h}_v^{(K)} | v \in V\}\!\})$$

We defined the $\overrightarrow{\text{READOUT}}$ function over multisets in analogy with the termination condition of the Weisfeiler-Leman test (see Equation (A.1) in Appendix A.1.3) in order to later compare MPNNs' expressive power to that of the Weisfeiler-Leman kernel.

We next see the most famous MPNN architectures from the literature. In the following, σ will be a non-linearity (ReLU, sigmoid, etc) and $W^{(k)}$ matrices of trainable weights

GraphSAGE

It is given by:

$$\begin{cases} \overrightarrow{\text{MSG}}_{u \to v}^{(k)} & = \text{id}(\cdot) \\ \overrightarrow{\text{AGGREGATE}}_v^{(k)} & = \begin{cases} MEAN \\ MAX \\ LSTM \end{cases} \\ \overrightarrow{\text{COMBINE}}^{(k)} & = \sigma(W^{(k)} \cdot) \end{cases}$$

Where the $LSTM$ aggregator is actually trained on a sample of all permutations of the message multiset, in order to achieve some degree of permutation-invariance. Notably, the [16] paper was the first to introduce *Neighbor Sampling* (see Section 2.3.1), and the L^2 normalization of intermediate embeddings. They test their architectures against node classification tasks.

Graph Attention Network (GAT)

Developed by [26], which builds on the idea that (self-)attention alone is capable of learning useful representation of sequence-based inputs [19]. We define the *attention coefficients*:

$$e_{uv}^{(k)} = a^{(k)}(W_{att}^{(k)} \vec{h}_u^{(k-1)}, W_{att}^{(k)} \vec{h}_v^{(k-1)})$$

Where a is a one-layer MLP:

$$a^{(k)}(W_{att}^{(k)}\vec{h}_u^{(k)}, W_{att}^{(k)}\vec{h}_v^{(k)}) = \text{LeakyReLU}(\vec{a}^T[W_{att}^{(k)}\vec{h}_u^{(k)}||W_{att}^{(k)}\vec{h}_v^{(k)}])$$

and $[\cdot||\cdot]$ represents concatenation. Furthermore, given their softmax-normalization:

$$\alpha_{uv}^{(k)} = \frac{\exp(e_{uv}^{(k)})}{\sum_{n\in N(v)} \exp(e_{nv}^{(k)})}$$

The GAT architecture corresponds to the choices:

$$\begin{cases} \overrightarrow{\text{MSG}}_{u\to v}^{(k)} & = \alpha_{uv}^{(k)} W^{(k)} \vec{h}_u^{(k-1)} \\ \overrightarrow{\text{AGGREGATE}}_v^{(k)} & = \sum_{u\in N(v)} \\ \overrightarrow{\text{COMBINE}}^{(k)} & = \sigma(\cdot) \end{cases}$$

The authors extend their architecture to include *multi-head attention*, that consists in training multiple attention heads and finally averaging them at *aggregation-time*:

$$\begin{cases} \overrightarrow{\text{MSG}}_{u\to v}^{(k)} & = \left\{ \alpha_{uv,j}^{(k)} W^{(k)} \vec{h}_u^{(k-1)} \right\}_{j=1}^{J} \\ \overrightarrow{\text{AGGREGATE}}_v^{(k)} & = \frac{1}{J} \sum_{j=1}^{J} \sum_{u\in N(v)} \end{cases}$$

The authors demonstrate the capabilities of this MPNN via node classification tasks.

Graph Isomorphism Network (GIN)

First developed in [36], the authors set out to identify the *most expressive* MPNN. In this context, *expressivity* is related to the number of non-isomorphic graphs a coloring/embedding algorithm is actually able to distinguish, given a $\overrightarrow{\text{READOUT}}$ function injective over multisets. By looking at the MPNN's aggregation strategy in Algorithm 1, we notice its analogy with the Weisfeiler-Leman kernel as described in section A.1.3. In particular,

MPNNs can be seen as a version of the Weisfeiler-Leman algorithm capable of acting on continuous node features and where the $HASH$ function has been substituted by a differentiable aggregation strategy given by a combination of the $\overrightarrow{\text{MSG}}^{(\cdot)}$, $\overrightarrow{\text{AGGREGATE}}^{(\cdot)}$ and $\overrightarrow{\text{COMBINE}}^{(\cdot)}$ methods, to enable training.

Recalling that the expressive power of the Weisfeiler-Leman algorithm resides in the injectivity of the $HASH$ over multisets, and since it could be that an MPNN's aggregation strategy is no longer injective, we conclude that MPNNs' expressivity is upper bounded by the Weisfeiler-Leman algorithm. For us, it means that if the Weisfeiler-Leman algorithm cannot distinguish between two graphs, then whatever MPNN won't be able to assign nodes embeddings to their nodes such that a subsequent multiset-injective $\overrightarrow{\text{READOUT}}$ may distinguish between the two graphs. Therefore, the expressive power of MPNNs is upper-bounded by the Weisfeiler-Leman test.

But can we design an MPNN such that, within the set of functions it may represent, some are as expressive as the Weisfeiler-Leman? Given what we just observed, it would suffice to find an MPNN whose aggregation strategy is multiset injective.

To this end, we state the Kolmogorov-Arnold Representation Theorem (Wikipedia):

Theorem 2.1.1 (Kolmogorov-Arnold) *Given a multivariate continuous function* $f : [0,1]^n \to \mathbb{R}$*, it can always be written as*

$$f(x_1, ..., x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_{q,p}(x_p) \right)$$

for some Φ_q *and* $\phi_{q,p}$*.*

It follows that if f is permutation-invariant *and* $x_1, ..., x_n$ come from a countable set $\mathcal{X}$, then Theorem 2.1.1 assumes the form [20], [36]:

$$f(x_1, ..., x_n) = \rho(\sum_{i=1}^{n} \phi(x_i)) \tag{2.2}$$

Let us prove it in the case where f is multiset-injective, which is the case we are interested in. If $\mathcal{X}$ is countable, then there exists a bijection $c : \mathcal{X} \to \mathbb{N}$, thus $\phi(x) = n^{-c(x)}$ is unique for each x and for each cardinality

of the multiset $\{\!\{x_1, ..., x_n\}\!\}$. Therefore, there always exists a ρ such that $f(x_1, ..., x_n) = \rho(\sum_{i=1}^{n} \phi(x_i))$ QED.

Therefore, if we come up with an *learnable* aggregation strategy such that it contains all functions of the form $\rho(\sum_{x \in X} \phi(x))$ over multisets $X \preccurlyeq \mathcal{X}$ ($\preccurlyeq$ is the multiset inclusion), then we are assured it may, in principle, learn an injective one and thus be as expressive as the Weisfeiler-Leman algorithm. To this end, [36] proposes:

$$\vec{h}_v^{(k)} = \text{MLP}^{(k)} \left((1 + \epsilon) \vec{h}_v^{(k-1)} + \sum_{u \in N(v)} \vec{h}_u^{(k-1)} \right) \tag{2.3}$$

Where the $(1 + \epsilon)\cdot$ term is added to make the aggregation strategy injective over the set of elements of the form (c, X) where $c \in \mathcal{X} \text{and} X \preccurlyeq \mathcal{X}$. Moreover, we modelled $\rho \circ \phi$ using one single MLP at each layer. Equation (2.3) is known as *Graph Isomorphism Network*.

As apparent also from the references, these results are only proven in the case where $\mathcal{X}$ is countable, leaving the question open when the nodes' features are continuous. As a little extension, we argue that since Theorem 2.1.1, for symmetric functions over continuous variables, reduces to:

$$f(x_1, ..., x_n) = \sum_{q=0}^{2n} \Phi_q \left(\sum_{p=1}^{n} \phi_q(x_p) \right)$$

Then a suitable aggregation strategy could be:

$$\vec{h}_v^{(k)} = \sum_{q=1}^{2n_{k-1}} \vec{\Phi}_q^{(k)} \left((1 + \epsilon) \vec{\phi}_q^{(k)}(\vec{h}_v^{(k-1)}) + \sum_{u \in N(v)} \vec{\phi}_q^{(k)}(\vec{h}_u^{(k-1)}) \right)$$

Where now $\vec{\Phi}_q^{(k)}$ and $\vec{\phi}_q^{(k)}$ are MLPs and n_k is the dimension of the k-th order embedding. This aggregation strategy is clearly very much prone to overfitting, so we believe this could be a valuable line for future research.

For more details about the proofs and mathematical background, please refer to [20] and [36].

2.1.2 Spectral Graph Convolutional Neural Networks (SpGCNs)

SpGCNs are another class of deep architectures over graphs, and in many instances are a special case of MPNNs. Nonetheless, they make the connection between Spectral Graph Theory ([73]) and Graph Representation Learning explicit, so we decided to briefly present them too.

The starting intuition is that we want to make the concept of "filter sliding over a graph" introduced in Section 2.1.1 rigorous. This amounts to defining a "convolution" operator over graphs, whose filter will be learnable in order to extract only task-relevant information at each position of the filter. We will develop graph convolution via an analogy with the continuous case.

Convolutions over Graphs

In the continuous uni-dimensional case, given a function f and a filter g (another function), their convolution is a third function $f * g$ given by:

$$(f * g)(y) = \int_{-\infty}^{+\infty} f(x)g(y-x)dx$$

It is widely known that convolutions are reduced to products under a Fourier transform. Given a function f, its Fourier transform $\mathcal{F}[f](\omega)$ is given by:

$$\mathcal{F}[f](\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} f(x)e^{-i\omega x}dx$$

And its inverse is:

$$f(x) = \mathcal{F}^{-1}[\mathcal{F}[f]](x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{+\infty} \mathcal{F}[f](\omega)e^{-i\omega x}d\omega$$

One can prove that:

$$\mathcal{F}[f * g](\omega) = \mathcal{F}[f](\omega)\mathcal{F}[g](\omega)$$

Which allows us to define the convolution of f and g as:

$$(f * g)(y) := \mathcal{F}^{-1}[\mathcal{F}[f](\cdot)\mathcal{F}[g](\cdot)](y) \tag{2.4}$$

Therefore, if we manage to define the Fourier transform over a graph, then we can also define a convolution over a graph by applying Equation (2.4).

We restrict this dissertation to undirected graphs. In order to define a discrete Fourier transform, we notice that in the definition of the continuous Fourier transform, the function f being transformed is integrated over all the eigenfunctions $e^{-i\omega x}$ of the continuous laplacian Δ. Thus, the problem of defining a discrete Fourier transform is reduced to that of defining a discrete laplacian. This issue is quickly resolved by noting that the matrix defined by:

$$L = D - A$$

is the discrete analogue of the Laplace operator, meaning that given a function $f : V \to \mathbb{R}$ such that $\vec{f} = (f(i))_{i=1}^{|V|}$ (a *signal*), then $L\vec{f}[i]$ is proportional to the difference between $f(i)$ and the average of $\{f(j)|j \in N(i)\}$. See e.g. this blog post. A similar intuition may be obtained via finite differences (Wikipedia, Medium). Note that L is symmetric, therefore it is diagonalizable and admits a complete set of orthogonal eigenvectors.

Thus, given a graph $G = (V, E)$, starting from its laplacian matrix L we can consider its orthogonal matrix of eigenvectors U. Analogously to the continuous case, given a function $f : V \to \mathbb{R}$ such that $\vec{f} = (f(i))_{i=1}^{|V|}$, we define its (discrete) Fourier transform as:

$$\hat{\vec{f}} := \mathcal{F}[\vec{f}] = U^t \vec{f}$$

And its inverse:

$$\mathcal{F}^{-1}[\mathcal{F}[\vec{f}]] = U\mathcal{F}[\vec{f}] \equiv U\hat{\vec{f}}$$

Which of course equals $\vec{f}$ due to U's orthogonality.

Also, given the Fourier transform $\hat{\vec{g}}_{\vec{\theta}}$ of a parametric filter $\vec{g}_{\vec{\theta}} = \text{diag}(\vec{\theta})$, we can use the above definition of discrete Fourier transform to finally define the convolution over graphs:

$$\vec{g}_{\vec{\theta}} * \vec{f} = U \hat{\vec{g}}_{\vec{\theta}} U^t \vec{f} \tag{2.5}$$

Graph Convolutional Network (GCN)

Starting from Equation (2.5), we may describe the most famous spectral graph convolutional network, which doubles as one of the first deep architecture over graphs: the *Graph Convolutional Network* (*GCN*) from [17].

For computational reasons, we will be working with the self-looped and normalized versions of A and L, meaning:

$$\begin{cases} \tilde{A} & = A + \mathbb{I} \\ \tilde{D} & = \begin{cases} \tilde{D}_{ii} & = \sum_{i=1}^{n} \tilde{A}_{ii} \\ \tilde{D}_{ij} & = 0 \text{ for } i \neq j \end{cases} \\ \hat{A} & = D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \\ \hat{L} & = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = \mathbb{I} - \hat{A} \end{cases} \tag{2.6}$$

It follows that also $\hat{L}$ is diagonalizable, and we will denote with $\hat{\Lambda}$ the diagonal matrix of its eigenvalues and with U the orthogonal matrix of its eigenvectors.

If we define the Chebishev polynomials:

$$\begin{cases} T_0(x) & = 1 \\ T_1(x) & = x \\ T_k(x) & = 2xT_{k-1}(x) - T_{k-2}(x) \end{cases} \tag{2.7}$$

Then we may interpret $\vec{g}_{\vec{\theta}}$ as a function of $\tilde{\Lambda} = \frac{2}{\lambda_{max}} \hat{\Lambda} - \mathbb{I}$ (see [4]):

$$\vec{g}_{\vec{\theta}} \approx \sum_{k=0}^{K} \theta_k T_k(\vec{\tilde{\Lambda}}) \tag{2.8}$$

Where $\vec{\tilde{\Lambda}}$ is the diagonal of $\hat{\Lambda}$. Substituting Equation (2.8)in the right hand side of Equation (2.5) after truncating the summation to the first order yields:

$$\vec{g}_{\vec{\theta}} * \vec{f} \approx U(\theta_0 \mathbb{I} + \theta_1 \tilde{\Lambda}) U^T \vec{f} = \theta_0 \vec{f} + \theta_1 U \tilde{\Lambda} U^T \vec{f} = (\theta_0 \mathbb{I} + \theta_1 \tilde{L}) \vec{f}$$

Where $\tilde{L} = \frac{2}{\lambda_{max}}\hat{L} - \mathbb{I}$

Assuming $\theta_0 = -\theta_1 =: \theta$ to reduce the number of parameters, and setting $\lambda_{max} = 2$ (the neural network may eventually *learn around* it), we get:

$$\vec{g_{\vec{\theta}}} * \vec{f} \approx \theta(\mathbb{I} - \hat{L} + \mathbb{I})\vec{f} = \theta(\mathbb{I} - \mathbb{I} + \hat{A} + \mathbb{I})\vec{f} = \theta(\mathbb{I} + \hat{A})\vec{f}$$

Due to the instabilities that might stem from the fact that $(\mathbb{I} + \hat{A})$ has eigenvalues comprised between 0 and 2, we apply the *renormalization trick* $\mathbb{I} + \hat{D} \to \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} := \hat{\tilde{A}}$. Thus we get:

$$\vec{g_{\vec{\theta}}} * \vec{f} \approx \theta\hat{\tilde{A}}\vec{f}$$

Considering $C \in \mathbb{N}$ node features i.e. multiple signals and $F \in \mathbb{N}$ filters:

$$H = \hat{\tilde{A}}X\Theta$$

Where $X \in \mathbb{R}^{N\times C}$ is the node feature matrix, and $\Theta \in \mathbb{R}^{C\times F}$. To increase expressivity, we may add a nonlinearity and repeat the application of the same operator several times:

$$H^{(k)} = \sigma(\hat{\tilde{A}}H^{(k-1)}\Theta^{(k)}) \tag{2.9}$$

Where $H^{(0)} = X$. Equation (2.9) is known as the Graph Convolutional Network [17].

The reader will have noticed by now that the operation we derived is equivalent to a very simple MPNN. Nevertheless, this discussion allowed us to more solidly ground the message passing operation as a generalization of an approximation to graph convolutions obtained by truncating Chebishev polynomial expansion of the filter. More sophisticated SpGCNs exist, such as [13], [33], [22] for undirected graphs, and e.g. [46], [45], [52] for directed graphs.

Since almost all SpGCNs can be reduced to MPNNs, in the remainder of this thesis we will use the *Graph Neural Network* (*GNN*) terminology to refer to both.

2.2 Graph Autoencoders

Graph Autoencoders are a family of models capable of learning a latent representation of every node in a graph, and therefore of all of its edges and also of the entire graph itself. Drawing inspiration from standard Autoencoders, they come in Simple and Variational Forms [12]. In either case, they follow the encoder-decoder paradigm where the decoder usually performs Link Prediction tasks. In this configuration, the Variational form also constitutes a very early attempt at synthetic data generation and is therefore tested in different forms and on different datasets in Section 2.5.3 and in Section 2.5.4. In the following, we will introduce Simple and Variational Autoencoders (Section 2.2.1) before describing their graph-adapted versions (Section 2.2.4 and Section 2.2.5).

2.2.1 Autoencoders

Autoencoders are a family of neural networks capable of performing unsupervised learning on a dataset with the purpose to encode and decode its latent features. "Features" is here taken in the broad sense, where it might refer either to a datapoint-specific embedding or to the learnt parameters of a distribution chosen to model the dataset's unknown true distribution.

The datapoint embedding-based approaches are usually referred to as the "standard" autoencoders (which in turn exist in many forms: denoising, contractive, under/overcomplete, etc), while deep neural networks aimed at learning the underlying distribution of a given dataset are known as Variational Autoencoders ([8]).

Any autoencoder is (at least conceptually) made of two components: an *encoder* and a *decoder*. The encoder takes the datapoints as inputs and embeds them to (usually) a lower dimensionality through a sequence of neural network layer. The final, deepest representation the encoder produces is known what we call a *latent representation*. The decoder takes this latent representation as input and attempts to reproduce the original input. Similarity metrics (like the *reconstruction error*) computed between the reconstructed output and the encoder's input are evaluated and used as loss function to train the whole autoencoder.

Both standard and variational approaches have been extended to graphs (see [12]). Before describing the graph-specific variants, we will briefly introduce

general-purpose autoencoders and variational autoencoders.

2.2.2 Simple Autoencoders

As noted previously, autoencoders are given by an encoder *Enc* and a decoder *Dec*. The purpose of the encoder is, given a feature matrix X, to compute a latent representation

$$H = Enc(X)$$

The decoder tries instead to reconstruct the original input. Thus, it will compute

$$\hat{X} = Dec(H)$$

If the composition $Dec \odot Enc$ is differentiable, then the entire architecture may be trained via backpropagation. In fact, both *Enc* and *Dec* could be neural networks.

2.2.3 Variational Autoencoders

The other family of autoencoders is given by variational autoencoders [8]. These architectures use the latent representations learned by the encoder to predict the parameters (location, shape) of a distribution from which to sample the predicted reconstructions. These models tend to display better performance when compared to standard autoencoders, at the expense of higher computational and conceptual complexity. In the following, we will discuss the details of a "vanilla" variational autoencoder (VAE).

Intuition and Motivation

The idea behind VAEs is to simultaneously learn both the underlying distribution $p^*(x)$ of a dataset $D = \{\vec{x}\}_{i=1}^{N}$, and a way to efficiently (with respect to the computation of expectation) sample from it datapoints similar to a given one.

This is done under the usual assumption that there exist a lower dimensional space of datapoints' representations that we can use to sample encoded features to be then "unpacked" by a decoder. This means that we are going to model $p^*(x)$ with two probability distributions: a $p(\vec{z}; \Theta)$which samples lower dimensional embeddings $\vec{z}$ | and another distribution $p(\vec{x}; \vec{f}(\vec{z}, \Theta))$which in turn samples datapoints $\vec{x}$ that we want to be "similar" to those found in D.

$\vec{f}$ can be a neural network that maps $\vec{z}$and Θ to the parameters of the distribution $p(\vec{x}; \vec{f}(\vec{z}, \Theta))$(e.g. its mean, variance, etc), while Θ are the parameters of $\vec{f}$ that will be learnt from D in order to maximize:

$$p(D|\Theta) = \sum_{x \in D} \int p(\vec{x}; \vec{f}(\vec{z}, \Theta)) p(\vec{z}; \Theta) dz \tag{2.10}$$

To clarify ideas and in order to obtain computational tractability, one may set $p(\vec{z}; \Theta)$to a distribution which is easy to sample and that is independent of Θ, e.g. $\mathcal{N}(\vec{0}, I)$, so that Equation (2.10) reduces to:

$$p(D|\Theta) = \sum_{x \in D} \int p(\vec{x}; \vec{f}(\vec{z}, \Theta)) \mathcal{N}(\vec{0}, I) dz$$

In fact, it can be proven that there always exists a transformation $\vec{f}$ that maps a simple distribution to an arbitrary one (see [11]) . The maximization of the above formula is what is usually called "maximum likelihood" i.e. We want to find Θ such that our model predicts as high a probability as possible for all points in D. Once the model has been trained, given a $\vec{z}$ sampled from $\mathcal{N}(\vec{0}, I)$, the probability of $\vec{x} \sim p(\vec{x}; \vec{f}(\vec{z}, \Theta))$ to be "similar" to those found in D will be maximized. And so this is how the model will generate new $\vec{x}$.

We could make an example. Consider a model that learns to sample (generate) $n \times n$ pictures of hand-written digits $D = \{x\}_{i=1}^{N}$. It is likely that there exists a lower dimensional embedding of hand-written digits where perhaps pictures that represent the same digit are closer together. Every time a lower dimensional representation is sampled by $p(\vec{z}; \Theta)$, the $p(\vec{x}; \vec{f}(\vec{z}, \Theta))$samples datapoint which, with high probability, is similar to those present in D.

The "autoencoder" flavor of this model will be made more evident below (actually, it will stem out of the necessity to obtain a differentiable computational graph), just like the need for the "variational" part. In the next section, we will describe the Variational Autoencoder in more detail, discussing its mathematical formulation and training procedure.

Description of the VAE Model and its Training Procedure

Our goal is to model an unknown probability distribution $p^*(x)$ from which nature has sampled $D = \{x\}_{i=1}^{N}$ in such a way that we will later be able to draw more samples with probability similar to $p^*(x)$.

We intend to model $p^*(x)$ via a parametric latent variable model:

$$p(\vec{x}, \vec{z}; \Theta) = p(\vec{x}|\vec{z}; \Theta)p(\vec{z}; \Theta) = p(\vec{x}; \vec{f}(\vec{z}, \Theta))p(\vec{z}; \Theta)$$

Where "|" indicates conditioning while ";" divides random variables from model parameters.

Thus, we wish to maximize the marginal likelihood:

$$p(D; \Theta) = \sum_{\vec{x} \in D} p(\vec{x}; \Theta) = \sum_{\vec{x} \in D} \int p(\vec{x}, \vec{z}; \Theta) d\vec{z} = \sum_{\vec{x} \in D} \int p(\vec{x}; \vec{f}(\vec{z}, \Theta)) p(\vec{z}; \Theta) d\vec{z}$$

With respect to Θ. Maximizing the above integral may be intractable, both analytically and/or numerically:

1. It may be analytically intractable since we may not know the primitive of the integral;
2. It may be numerically intractable since a numerical approximation could require to sample M values $\{\vec{z}_i\}_{i=1}^M$ from $p(\vec{z}; \Theta)$, and then compute:

$$p(D; \Theta) \approx \sum_{\vec{x} \in D} \sum_{\vec{z}_i \in \{\vec{z}_i\}_{i=1}^M} p(\vec{x}; \vec{f}(\vec{z}_i, \Theta))$$

But it could be the case where too large a sample of $\{\vec{z}_i\}_{i=1}^M$ is required that exceeds our computational availabilities. That is, it could be that $p(\vec{z}; \Theta)$is not a good approximation of $p(\vec{z}|\vec{x}; \Theta)$and so it may take many samples from $p(\vec{z}; \Theta)$to properly approximate the marginal likelihood $p(\vec{x}; \vec{f}(\vec{z}, \Theta))$.

In order to circumvent these issues, we are going to learn from D together with $\vec{f}(\vec{z}, \Theta)$, also a distribution $q(\vec{z}; \vec{g}(\vec{x}, \Phi))$(where $\vec{g}(\vec{x}, \Phi)$could be a neural network parametrized by the learnable Φ). The role of q is to learn $p(\vec{z}|\vec{x}; \Theta)$, in order to efficiently approximate the marginal likelihood $p(\vec{x}; \vec{f}(\vec{z}, \Theta))$.

This opens the question on which objective should we maximize. We are clearly not satisfied with the maximum likelihood above, since the dependency on q is not made explicit and as we will see, its explicitation will highlight a whole set of issues that require our attention.

We may indeed look for such objective by making the marginal likelihood's dependence on q explicit. We will compactify the notation as follows:

$$\begin{cases} p_\Theta(\vec{z}) := p(\vec{z};\Theta) \\ p_\Theta(\vec{x}) := p(\vec{x};\Theta) \\ p_\Theta(\vec{z}|\vec{x}) := p(\vec{z}|\vec{x};\Theta) \\ p_\Theta(\vec{x},\vec{z}) := p(\vec{x},\vec{z};\Theta) \\ q_\Phi(\vec{z}|\vec{x}) := q(\vec{z};\vec{g}(\vec{x},\Phi)) \end{cases} . \tag{2.11}$$

With this notation, we observe that (note that $p_\Theta(\vec{x})$ does not depend on $\vec{z}$):

$$\begin{aligned} \ln p_\Theta(\vec{x}) &= E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln p_\Theta(\vec{x})\right] = E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln \frac{p_\Theta(\vec{x},\vec{z})}{p(\vec{z}|\vec{x};\Theta)}\right] = E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln\left(\frac{p_\Theta(\vec{x},\vec{z})}{q_\Phi(\vec{z}|\vec{x})}\frac{q_\Phi(\vec{z}|\vec{x})}{p_\Theta(\vec{z}|\vec{x})}\right)\right] \\ &= \underbrace{E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln \frac{p_\Theta(\vec{x},\vec{z})}{q_\Phi(\vec{z}|\vec{x})}\right]}_{=:\mathcal{L}_{\Theta,\Phi}(\vec{x})} + \underbrace{E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln \frac{q_\Phi(\vec{z}|\vec{x})}{p_\Theta(\vec{z}|\vec{x})}\right]}_{\equiv D_{KL}(q_\Phi(\vec{z}|\vec{x})||p_\Theta(\vec{z}|\vec{x}))} \end{aligned} \tag{2.12}$$

Where we recognize that the second term is a Kullback-Leibler Divergence while the first is known as the *Variational Lower Bound* or *Evidence Lower Bound* (ELBO), and by definition it equals:

$$\mathcal{L}_{\Theta,\Phi}(\vec{x}) \equiv \ln p_\Theta(\vec{x}) - D_{KL}\left(q_\Phi(\vec{z}|\vec{x})||p_\Theta(\vec{z}|\vec{x})\right) \tag{2.13}$$

Or equivalently:

$$\begin{aligned} \mathcal{L}_{\Theta,\Phi}(\vec{x}) &\equiv E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln p_\Theta(\vec{x},\vec{z}) - \ln q_\Phi(\vec{z}|\vec{x})\right] \\ &= E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln\left(p_\Theta(\vec{x})\frac{p_\Theta(\vec{x}|\vec{z})p_\Theta(\vec{z})}{p_\Theta(\vec{x})}\frac{1}{q_\Phi(\vec{z}|\vec{x})}\right)\right] \\ &= E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln\left(\frac{p_\Theta(\vec{x}|\vec{z})p_\Theta(\vec{z})}{q_\Phi(\vec{z}|\vec{x})}\right)\right] \\ &= -D_{KL}\left(q_\Phi(\vec{z}|\vec{x})||p_\Theta(\vec{z})\right) + E_{q_\Phi(\vec{z}|\vec{x})}\left[\ln p_\Theta(\vec{x}|\vec{z})\right] \end{aligned} \tag{2.14}$$

Thus, looking at Equation (2.13), maximizing the ELBO corresponds to simultaneously maximize the marginal likelihood and minimizing the divergence between $q_\Phi(\vec{z}|\vec{x})$and $p_\Theta(\vec{z}|\vec{x})$(i.e. pulling $q_\Phi(\vec{z}|\vec{x})$towards being an efficient sampler of $p(\vec{x}; \vec{f}(\vec{z}, \Theta))$). Since both effects are desirable, the ELBO is usually chosen as the loss for these models.

There are, however, a few issues with this approach.

If we were to use the ELBO as the loss for a stochastic gradient descent algorithm (as is often the case), it would mean that at every time a forward pass with a given batch $M \subset D$ is completed, parameters would be updated according to:

$$\Theta, \Phi \longleftarrow \Theta, \Phi - \eta \nabla_{\Theta,\Phi} \left(E_{\vec{x}\sim M} \left[E_{q_\Phi(\vec{z}|\vec{x})} \left[\ln p_\Theta(\vec{x}, \vec{z}) - \ln q_\Phi(\vec{z}|\vec{x}) \right] \right] \right)$$

Note that we can get an unbiased estimator for ∇_Θ:

$$\begin{aligned} &\nabla_\Theta \left(E_{\vec{x}\sim M} \left[E_{q_\Phi(\vec{z}|\vec{x})} \left[\ln p_\Theta(\vec{x}, \vec{z}) - \ln q_\Phi(\vec{z}|\vec{x}) \right] \right] \right) \\ &= E_{\vec{x}\sim M} \left[E_{q_\Phi(\vec{z}|\vec{x})} \left[\nabla_\Theta \left(\ln p_\Theta(\vec{x}, \vec{z}) - \ln q_\Phi(\vec{z}|\vec{x}) \right) \right] \right] \\ &\simeq E_{\vec{x}\sim M} \left[\nabla_\Theta \left(\ln p_\Theta(\vec{x}, \vec{z}) - \ln q_\Phi(\vec{z}|\vec{x}) \right) \right] \end{aligned} \tag{2.15}$$

Where the last $\simeq$ is due to a Monte Carlo approximation. But the second equality does not hold for ∇_Φ, which would force us to compute the primitive of $E_{q_\Phi(\vec{z}|\vec{x})} [\ldots]$ every time, and this could make the optimization intractable.

Moreover, the computational model we developed so far does not admit backpropagation. In fact, the training-time forward pass would be:

1. Sample $\vec{x} \sim D$;
2. Compute $\vec{g}(\vec{x}, \Phi)$;
3. Sample $\vec{z} \sim q(\vec{z}; \vec{g}(\vec{x}, \Phi))$;
4. Compute $\vec{f}(\vec{z}, \Theta)$;
5. Compute the $\mathcal{L}_{\Theta,\Phi}(\vec{x})$.

But the "sampling" is not a differentiable operation, and thus libraries like Pytorch or TensorFlow would not be able to compute gradients.

These two reasons induce us to modify the model as follows. We introduce a new distribution $p(\vec{\epsilon})$from which we sample an $\vec{\epsilon}$ that we feed, together with $\Phi$and $\vec{x}$, to a function $\vec{s}(\vec{\epsilon},\Phi,\vec{x})$which produces a $\vec{z}$. So the the training-time forward pass becomes:

1. Sample $\vec{x} \sim D$ and $\vec{\epsilon} \sim p(\vec{\epsilon})$;
2. Compute $\vec{g}(\vec{x},\Phi)$;
3. Compute $\vec{z} = \vec{s}(\vec{\epsilon},\Phi,\vec{x})$;
4. Compute $\vec{f}(\vec{z},\Theta)$;
5. Compute the $\mathcal{L}_{\Theta,\Phi}(\vec{x})$.

Since we are no longer sampling in the middle of the forward pass (but only on leaf input/output nodes), the resulting computation graph admits backpropagation.

Of course, the functional form of $\vec{s}(\vec{\epsilon},\Phi,\vec{x})$that we choose should somewhat reproduce, given Φ, a sampling operation from $q(\vec{z};\vec{g}(\vec{x},\Phi))$. For instance, if

$$q(\vec{z};\vec{g}(\vec{x},\Phi)) = \mathcal{N}(\vec{z};\vec{\mu}(\vec{x},\Phi),\Sigma(\vec{x},\Phi))$$

Then $\vec{s}(\vec{\epsilon},\Phi,\vec{x})$could for example be:

$$\vec{s}(\vec{\epsilon},\Phi,\vec{x}) = \vec{\mu}(\vec{x},\Phi) + \Sigma(\vec{x},\Phi)\vec{\epsilon}$$

Given an arbitrary q, the reader interested in finding the joint optimal $\vec{s}(\vec{\epsilon},\Phi,\vec{x})$ and $p(\vec{\epsilon})$ is referred to section 2.4 of [8] .

Thus, given a minibatch $M \subset D$ and an hyperparameter $L \in \mathbb{N}^+$, the estimator of $\mathcal{L}_{\Theta,\Phi}(\vec{x})$ we will be using to compute gradients is (using Equation (2.13)):

$$\tilde{\mathcal{L}}^{A}_{\Theta,\Phi}(M) = \sum_{i=1}^{|M|}\sum_{l=1}^{L} \ln p_\Theta(\vec{x}_i,\vec{z}_{i,l}) - \ln q_\Phi(\vec{z}_{i,l}|\vec{x}_i)$$

Where

$$\vec{z}_{i,l} = \vec{s}(\vec{\epsilon}_l,\Phi,\vec{x}_i) \text{ and } \{\vec{\epsilon}\}_{l=1}^{L} \sim p(\vec{\epsilon})$$

Alternatively, if the analytical form of $D_{KL}\left(q_\Phi(\vec{z}|\vec{x})||p_\Theta(\vec{z})\right)$ is known, one may use 2.14 :

$$\tilde{\mathcal{L}}^B_{\Theta,\Phi}(M) = -D_{KL}\left(q_\Phi(\vec{z}|\vec{x}_i)||p_\Theta(\vec{z})\right) + \frac{1}{L}\sum_{i=1}^{|M|}\sum_{l=1}^{L}\ln p_\Theta(\vec{x}_i|\vec{z}_{i,l})$$

Where, as before:

$$\vec{z}_{i,l} = \vec{s}(\vec{\epsilon}_l, \Phi, \vec{x}_i) \text{ and } \{\vec{\epsilon}\}_{l=1}^{L} \sim p(\vec{\epsilon})$$

In practice, if $|M| > 100$, L can be set to 1 (see [8]).

For more information on better losses, interpretations and choices of functional forms, please refer to the cited sources [12] , [11] and [28] .

2.2.4 Graph Autoencoders (GAEs)

The encoder of GAE from [12] is constituted by a GCN layer that takes the adjacency matrix A and the node feature matrix X as inputs and outputs a latent representation:

$$Enc(A, X) = GCN(A, X) = ReLU(\tilde{A}XW_0)$$

Where:

- $\tilde{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the re-normalized adjacency matrix;
- W_0 is a matrix of learned parameters;
- the ReLU is applied element-wise.

Of course, any graph neural network GNN$(\vec{x}, A, \vec{\theta}, c, a)$n be used as an encoder to perform link prediction. If we denote the matrix of all embeddings by Z, then the decoder tries to reconstruct the adjacency matrix via:

$$\hat{A} = \sigma(ZZ^T)$$

And can be thus trained via the binary cross-entropy loss (see section 2.4.2).

2.2.5 Variational Graph Autoencoders (VGAEs)

Variational Autoencoders have been adapted to graphs to obtain generative models for their adjacency matrix ([12]).

If we name with $A = [\vec{a}_1, ...\vec{a}_N]$ the adjacency matrix, $X = [\vec{x}_1, ..., \vec{x}_N]$ the node feature matrix. We may define:

$$GCN_1(A, X) = ReLU(\tilde{A}XW_0) \text{ and } GCN_1(\vec{x}_i) := GCN_1(A, X)[:, i]$$

Where $\tilde{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the re-normalized adjacency matrix. Moreover we set:

$$\begin{aligned} GCN_2(\vec{x}) &:= \tilde{A}\, GCN_1(\vec{x})\, W_1 =: CONCAT(GCN_{\vec{\mu}}(\vec{x}), GCN_{\vec{\sigma}}(\vec{x})) \\ &=: CONCAT(\vec{\mu}(\vec{x}), \vec{\sigma}(\vec{x})) \end{aligned} \tag{2.16}$$

Thus, GCN_1 embeds each node, while GCN_2 computes the mean and the variance of a normal distribution $\mathcal{N}(\vec{z}; \vec{\mu}(\vec{x}), \vec{\sigma}(\vec{x}))$ that will be our $q_\Phi(\vec{z}|\vec{x})$. Then we recognize that W_0 and W_1 will collectively form Φ and :

$$\vec{g}(\vec{x}, \Phi) = GCN_2(\vec{x})$$

Since the nodes come in a countable fashion, also $q_\Phi(Z|A)$ is defined . Following [12] and [8] , we may implement $\vec{s}(\vec{\epsilon}, \Phi, \vec{x})$ as:

$$\vec{s}(\vec{\epsilon}, \Phi, \vec{x}) := \vec{\mu}(\vec{x}) + diag(\vec{\sigma}(\vec{x}))\vec{\epsilon}$$

Finally, if we wish to use the VGAE for link prediction, the likelihood of the model will be given by:

$$p_\Theta(A|Z) = p(A|Z) = \prod_{i=1}^{N}\prod_{j=1}^{N} p(A_{i,j}|Z) := \prod_{i=1}^{N}\prod_{j=1}^{N} \sigma(\vec{z}_i^T \vec{z}_j)$$

Where σ is the sigmoid. So there is actually no parameter Θ to train in this decoder. The only trainable parameters are those of the encoder Φ.

As a prior $p_\Theta(\vec{z})$ we choose a gaussian (again there is no dependence on Θ):

$$p_\Theta(\vec{z}) = p(\vec{z}) := \mathcal{N}(\vec{z}|0, I)$$

We choose the loss:

$$\tilde{\mathcal{L}}^B_\Phi(A) = -D_{KL}\left(q_\Phi(Z|A)||p(Z)\right) + E_{q_\Phi(Z|A)}\left[\ln p(A|Z)\right]$$

And so we have constructed a Variational Graph Autoencoder.

2.3 Scaling Up GNNs

GNNs' training is known to be faster when performed on GPUs. Anyway, due to the limited and costly availability of VRAM, it is crucial to develop new techniques to allow the application of GNNs to large graphs, which would otherwise not be able to fit on video memory.

We are going to describe one training technique here (*Neighbor Sampling*) and report about two performance-oriented GNN architectures (*(Advanced) Cluster GCNs* and *Simplified GCNs*) in Appendix B.

We will use Neighbor Sampling extensively in some of our early applications in Section 2.5.

2.3.1 Neighbor Sampling

The first iterations of GNNs were trained using full-batch gradient descent. This approach, due to its memory requirements, was destined to fail when the network size would reach the hundreds of million up to billions of nodes that we measure in modern networks e.g. in social networks and publicly available knowledge graphs.

Nonetheless, industrial needs (e.g. recommender systems on huge user-item networks like that of Amazon or Alibaba) still required the application of deep learning on large graphs, and thus more sophisticated training techniques were developed to meet this demand.

One of the most basic approaches is to substitute full batch GD with *Stochastic Gradient Descent* (*SGD*), where, instead of computing the embeddings of all N nodes, only a minibatch $M << N$ of them is considered at each time. The problem with this approach is that, especially if the graph is large, the

randomly-sampled M nodes could have totally disjoint neighborhoods and thus aggregation wouldn't take place and the computed gradient wouldn't be representative of that given by the full batch. But, as we argued before, it's often impossible to fit the full batch into video memory.

A compromise would be that of storing only the K-hop neighborhoods of each of the M nodes in memory. In fact, given a K-layer GNN, only the information coming from the K-hop neighborhood of a node is required to compute its embeddings. Then, we could just store in memory the K-layer computation graph of each of the M node.

This approach, although promising, is actually not enough when networks are really big and/or when hubs (i.e. nodes with high in-degree) are present. In fact, we may have an exponential explosion of neighbors every time we increase K by 1. To compensate, one may set to a fixed H the number of neighbors to sample from each node at each layer. Such H neighbors may be chosen at random (not recommended on scale-free networks with hubs) or depending on their in-degree/number of appearances in RW-based samplings.

The latter technique, first developed in [16], is known as *Neighbor Sampling*, and, although it does not prevent the exponential explosion with K, it mitigates it enough to allow the usage of GNNs on large networks. The authors of [16] use neighbor sampling to compute the forward propagation rule of the GraphSAGE architecture as in Figure 2.4.

2.4 Tasks over Graphs

In this section we will be reviewing common inference tasks over graphs. Given the nature of this thesis, we will describe all tasks having a deep learning setting in the back of our minds. These definitions will later be applied in concrete experiments in Section 2.5.

2.4.1 Node Classification

Given a graph $G = (V, E)$ with N vertices, its nodes may have $d^{(0)}$-dimensional features (usually described with a feature matrix $X \in \mathbb{R}^{N \times d^{(0)}}$) and categorical labels $L : V \supseteq V_L \to \mathbb{N}_L$ where $\mathbb{N}_L$ is a finite subset of $\mathbb{N}$. In the ideal case, $|V_L| = |V|$, but in many applications not all labels are known and we wish to predict the missing ones by exploiting the existing information about the labelled nodes' features-labels relations and the graph structure.

Algorithm 2: GraphSAGE minibatch forward propagation algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$;
input features $\{\mathbf{x}_v, \forall v \in \mathcal{B}\}$;
depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$;
non-linearity σ;
differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$;
neighborhood sampling functions, $\mathcal{N}_k : v \rightarrow 2^{\mathcal{V}}, \forall k \in \{1, ..., K\}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{B}$

$\mathcal{B}^K \leftarrow \mathcal{B}$;
for $k = K...1$ **do**
  $B^{k-1} \leftarrow \mathcal{B}^k$;
  for $u \in \mathcal{B}^k$ **do**
    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$;
  end
end
$\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$;
for $k = 1...K$ **do**
  for $u \in \mathcal{B}^k$ **do**
    $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\})$;
    $\mathbf{h}_u^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k)\right)$;
    $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$;
  end
end
$\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$

Figure 2.4: The forward computation of GraphSAGE. *Courtesy of [16]*

The usual way of performing a node classification tasks via a Graph Neural Network $\mathcal{M}_\Theta$ consists in splitting V_L in train (supervision) V_L^S, validation V_L^V and test V_L^T sets, and then training the GNN via backpropagation using, e.g., mean cross entropy loss:

$$\mathcal{L}_{CE} = -\frac{1}{\sum\limits_{i \in \mathbb{N}_L} w_i} \sum_{v \in V_L^S} w_{L(v)} \ln(\hat{p}(L(v)|v))$$

Where $\hat{p}(L(v)|v)$ is the probability of node v belonging to its actual class $L(v)$ as predicted by the model, and the w_is are optional class weights with $i \in \mathbb{N}_L$.

As test metrics, it is customary to use the micro-averaged F1-score:

$$F_1^{micro} = \frac{2 \sum\limits_{i \in \mathbb{N}_L} TP_i}{\sum\limits_{i \in \mathbb{N}_L} 2TP_i + FP_i + FN_i}$$

Which for *multiclass* (NOT *multilabel*) classification reduces to *accuracy*:

$$Acc = \frac{\sum\limits_{i \in C} TP_i + TN_i}{\sum\limits_{i \in C} TP_i + FP_i + TN_i + FN_i}$$

The difference between multiclass and multilabel classification is that in the former, class membership is exclusive.

2.4.2 Link Prediction

Link Prediction has been established as a foundational task within the Graph Representation Learning domain, both in the Academy and Industry. Various models have been devised throughout the years, ranging from Direct Inference approaches to Graph Deep Learning models. In this section, we will focus on Link Prediction [5], both Directed and Undirected. Link prediction has several applications, among which completing knowledge graphs ([7] and subsequent score functions), baseline approaches to deep graph generation [12], transaction network pre-processing [59] and recommender systems [3].

Concretely, given a graph $G = (V, E)$, Link Prediction consists in predicting missing links from E. Intuitively, we assume that links in E are are formed according to some (possibly ill-defined) rules that I'd like our algorithm to approximate. For the purposes of this thesis, we will restrict ourselves to Neural Link Prediction i.e. Link Prediction performed via GNNs using autoencoder models from Section 2.2 (but we will drop the "Neural" word).

Practically, there are two possible definitions of link prediction, that we name *Standard Link Prediction* and *Effective Link Prediction*. They differ in how E is split between training, validation and test sets.

Standard link Prediction

The *Standard* approach to Link Prediction prescribes to split E in four sets:

- *Message Edges* E_m;
- *Supervision (positive) Edges* E_s^p;
- *Validation (positive) Edges* E_v^p;
- *Test (positive) Edges* E_t^p.

We also sample sets of *negative* i.e. non-existent edges E_s^n, E_v^n and E_t^n. For clarity's sake, we won't make particular assumptions on how these edges are sampled right now, even though we will recognize this as crucial issue later on. We therefore define:

$$\begin{aligned} E_s &= E_s^p \cup E_s^n \\ E_v &= E_v^p \cup E_v^n \\ E_t &= E_t^p \cup E_t^n \end{aligned} \tag{2.17}$$

During training, message passing is performed over E_m, while E_s contribute to the training loss $\mathcal{L}_T$. During validation, $E_m \cup E_s$ are used for message passing, while only E_v contribute to the validation loss. During testing, $E_m \cup E_s \cup E_v$ are used for message passing, while E_t contribute to the test metric. Therefore, the training phase of Link Prediction reduces to a binary classification task, for which we may use binary cross entropy as loss:

$$\mathcal{L}_T = \sum_{e_{uv} \in E_s} p_c y_{uv} \ln(\hat{p}(e_{uv})) + (1 - y_{uv}) \ln(1 - \hat{p}(e_{uv}))$$

Where $y_{uv} = 1$ is the ground truth exists ($y_{uv} = 1$)/ doesn't exist ($y_{uv} = 0$) label for edge e_{uv}, and p_c is an optional re-weighting of the positive edges' contribution (might be necessary due to the usual sparsity of networks). $\hat{p}(e_{uv})$ is the probability that edge e_{uv} exists as predicted by the model.

Although conceptually sound, this *Standard* approach to Link Prediction has the downside of using very few edges for supervision, and a random sample of negative edges would always be far from including all possible negative edges. Therefore, it is has been empirically proved that having E_s and E_m coincide together with using all remaining edges as negatives during training is actually beneficial. This approach, that we will name *Effective Link Prediction* is described in the next section.

Effective Link Prediction

The *Effective* approach to Link Prediction prescribes to condense the *Standard*'s E_m and E_s^p sets in one E_{m+s}^p set that will simultaneously act as message and supervision edge set during training. Furthermore, all edges in $V \times V \backslash E_{m+s}^p$ are used as negatives during training. We will denote with E_{m+s} the full *Effective* training set. Validation and test sets are instead built as in the *Standard* approach.

We note that the *Effective* approach has the following conceptual issues:

1. The training task is *different* from the test and validation tasks: supervision and message edges are the same, while validation and test edges are not used for message passing;

2. The training set contains the validation and test positive edges as negatives, therefore teaching the "wrong thing" to the model.

Nevertheless, it is also true that:

1. E_{m+s} contains far more positive and negative edges when compared to the *Standard*'s E_s, allowing the model to extract much more information;

2. The *Effective* approach is more *realistic* when it comes to the choice of training negative edges. In fact, it just assumes that edges that are not observed at training time are non-existent.

Empirically, it is observed that the *Effective* approach leads to far better results (see below).

2.5 Reproductions and Early Contributions

In the following, we will detail the reproduction of node classification and link prediction tasks found in papers and official benchmarks. We will first review useful metrics that will be later used (Section 2.5.1) and then proceed with reproduction of experiments on node classification (Section 2.5.2), undirected and directed link prediction (Section 2.5.3 and Section 2.5.4). These tasks, especially those related to Directed Link Prediction, will set the stage for our first major contribution in Chapter 3.

2.5.1 Test Metrics

Given a test set $E_t = E_t^p \cup E_t^n$, the following test metrics are widely adopted in the Link Prediction literature.

Receiving Operating Characteristic-Area Under the Curve

Given the usual confusion matrix TP, FP, TN, FN the *True Positive Rate* (TPR) and the *False Positive Rate* (FPR) are defined by:

$$\begin{cases} TPR &= \frac{TP}{P} \\ FPR &= \frac{FP}{N} \end{cases} \tag{2.18}$$

In order to obtain a class prediction from a binary classifier, we also need to set a threshold $T \in [0,1]$. A perfect classifier would predict $(TPR, FPR) = (1,0) \forall T \in [0,1]$ while clearly, a tradeoff exists between TPR and FPR for imperfect classifiers. In fact, for $T \to 0$ imperfect classifier will usually yield $TPR \to 1$ and $FPR \to 1$, while $T \to 1$ usually implies $TPR \to 0$ and $FPR \to 0$. A random classifier that predicts $\hat{T}$ with probability ρ will have $TP = \rho P$ and $FP = \rho N \implies TPR = FPR$. Since the choice of threshold is arbitrary (or better yet, it can be tuned), and the two classes may be unbalanced, it is useful to develop a metric that:

1. Evaluates the consistency of the model's performance at different threshold values;
2. Is insensitive to class imbalance.

Such a metric can be developed by plotting the *Receiving Operating Characteristic* (ROC) curve $FPR(T)$ vs $TPR(T)$, and then computing the *Area*

Under the Curve (AUC) from 0 to 1 (using horizontal interpolation).

This metric clearly takes into account model's performance at different threshold values, and it is also insensitive to class imbalance since it combines one score per each class. Please refer to [2] for more information.

A perfect classifier would display $ROC\text{-}AUC = 1$, while a random classifier has $ROC\text{-}AUC = 0.5$.

In the context of Link Prediction, $ROC\text{-}AUC$ can be approximated using:

$$ROC\text{-}AUC = \frac{n' + 0.5n''}{n}$$

Where n' is the number of times a positive edge scores higher than a randomly-sampled negative edge, while n'' counts the times their scores are equal (see [62] and [65]).

Average Precision

We define *Precision* Pr and *Recall* Rec as:

$$\begin{cases} Pr &= \frac{\mathrm{TP}}{\hat{P}} \\ Rec &= \frac{TP}{P} \end{cases} \tag{2.19}$$

Of course, also Pr and Rec are functions of the threshold T, and so as before we obtain a threshold-independent metric by plotting the *Precision-Recall* curve $Pr(T)$ vs $Rec(T)$ curve and then computing its Area Under the Curve, $PR\text{-}AUC$.

We notice that since both Pr and Rec are mainly concerned with the positive class only, $PR\text{-}AUC$ is indeed affected by class imbalance.

Given $\mathbb{I}$ increasing threshold values, It can be operationally computed as (see Scikit-Learn):

$$PR\text{-}AUC = \sum_{i=1}^{I} (Rec(T_i) - Rec(T_{i-1}))Pr(T_i)$$

Hits@K

The Hits@K metric computes the fraction of times the model ranks a positive edge ($\in E_t^p$) within the first K negative edges in E_t^n. This is a much more challenging metric compared to *ROC-AUC* and *PR-AUC*. Notice that a random classifier would attain $Hits@K \approx 0$.

Mean Reciprocal Rank (MRR)

MRR computes the average of the reciprocal ranks of each positive test edge in E_t^p among N sampled negative edges with the same head. This is a much more challenging metric compared to *ROC-AUC* and *PR-AUC*, and constitutes a soft version of *Hits@K*. Notice that a random classifier would attain $MRR \approx 0$.

2.5.2 Node Classification Experiments

Reproduction of GraphSAGE paper

We reproduced [16] results on the Reddit dataset. The dataset is an undirected graph where nodes represent posts, and two nodes are linked if one user commented below both of them. Node features are given by 300-dimensional *word embedding* of the concatenation of the post's title, comments, score and number of comments. The categorical label to predict is the subreddit the post was delivered to, and there are 41 subreddits in total.

The models we tried are:

- 'GraphSAGE_GCN':

$$\vec{h}_v^{(k)} = \text{L2-NORMALIZE}\left(\sigma\left(W^{(k)}\text{MEAN}(\{\vec{h}_v^{(k-1)}\} \cup \{\vec{h}_u^{(k-1)} | u \in N(v)\})\right)\right)$$

- 'GraphSAGE_MEAN':

$$\vec{h}_v^{(k)} = \text{L2-NORMALIZE}\left(\sigma\left(W_N^{(k)}\text{MEAN}(\{\vec{h}_u^{(k-1)} | u \in N(v)\}) + W_S^{(k)}\vec{h}_v^{(k-1)}\right)\right)$$

Which were taken directly from the paper, and two more models:

- 'ModGIN':

Model	NSI (F_1^{micro})	FNI (F_1^{micro})	epochs	secs/epoch
GraphSAGE_GCN	0.941	0.947	5	16.44
GraphSAGE_GCN_Sp	0.941	0.948	10	*14.68*
GraphSAGE_MEAN	0.957	0.962	3	19.15
GraphSAGE_MEAN_Sp	0.960	0.963	*2*	15
ModGIN	*0.961*	0.7	25	25.22
ModGIN_Att_Norm	0.959	*0.964*	10	46.27

Table 2.1: Results of several models on the Reddit dataset.

$$\vec{h}_v^{(k)} = \text{MLP}_C\left(\text{CONCAT}\left(\sum_{u \in N(v)} \text{MLP}_N(\vec{h}_u^{(k-1)}), \vec{h}_v^{(k-1)}\right)\right)$$

- 'ModGIN_Att_Norm':

$$\tilde{\vec{h}}_v^{(k)} = \text{MLP}_C\left(\left(\sum_{u \in N(v)} \text{MLP}_N(\vec{h}_u^{(k-1)} || \vec{h}_v^{(k-1)})\right) || \vec{h}_v^{(k-1)}\right)$$

We developed 'ModGIN' and 'ModGIN_Att_Norm' with the aim to improve upon the Graph Isomorphism Network in Section 2.1.1.

We managed to reproduce all results, and our models slightly outperformed the authors':

Due to the dimension of the dataset (232'965 total nodes, 492 average degree), we implemented the Neighbor Sampling technique. We obtained the results in Table 2.1, which are within the error bars reported in the paper [16].

Where *NSI* (Neighbor Sampling Inference) F_1^{micro} performance on the test set using the same Neighbor Sampling employed during training, while *FNI* (Full Neighborhood Inference) reconfigures GNN computations to take advantage of the full neighborhood of each node during testing. The latter technique is not compatible with training since it requires to keep in memory only the k-th order embeddings of each node before computing the $k+1$-th order embeddings, thus erasing part of the gradients needed for backpropagation (see code). 'Sp' stands for "sparse" (see Pytorch Geometric documentation). The hyperparameters we used are:

- 10 epochs, 512 batch size

- $K = 2$
- $S_1 = 25$, $S_2 = 10$ (neighbors sampled at each hop)
- Embedding dimension: 256 (big), 128 (small)
- $\sigma =$ ReLU
- Adam Optimizer, LR Hyperparameter Search: $\{0.01, 0.001, 0.0001\}$

We also tried using a training loss closer to the test metric, namely the differentiable micro-F1 (see [[9]]):

$$\begin{cases} TP = \sum\limits_{i \in C} o_i t_i \\ FP = \sum\limits_{i \in C} o_i (1 - t_i) \\ FN = \sum\limits_{i \in C} (1 - o_i) t_i \end{cases} \tag{2.20}$$

But it yielded no significant improvements.

Reproduction of OGB-Team attempt on OGBN-Arxiv dataset

OGB is a Stanford-run effort where several Graph Neural Networks are benchmarked against several datasets across node classification, link prediction and graph classification tasks (as of writing). We tried to reproduce the performance obtained by a GraphSAGE implementation over a node classification task on the OGBN-Arxiv dataset. The dataset is a directed citation networks where nodes represent papers, and two nodes are linked if one cites the other. Node features are given by 300-dimensional *word embedding* of the concatenation of the post's title, comments, score and number of comments. The categorical label to predict is the subreddit the post was delivered to, and there are 41 subreddits in total. Node features are obtained via averaging the embeddings of their titles and abstracts, resulting in 128-dimensional vectors. There are 169343 nodes and 1166243 edges. The split is as follows: articles published before 2017 (included) constitute the training set, 2018 makes the validation set, while 2019 articles are the test set.

As per models, we used:

- 'GraphSAGE_MEAN_OTS':

$$\vec{h}_v^{(k)} = W_N^{(k)} \text{MEAN}(\{\vec{h}_u^{(k-1)} | u \in N(v)\}) + W_S^{(k)} \vec{h}_v^{(k-1)}$$

Model	Accuracy	epochs	secs/epoch
GraphSAGE_MEAN_OTS	*0.7164*	154	*0.12*
GraphSAGE_MEAN_MB_NS	0.704	*10*	6.64

Table 2.2: Results obtained over the OGBN-Arxiv datasets.

The network is made of three such layers. The first two have L2-batch normalization after aggregation followed by a 'ReLU' activation and optional dropout. The third layer is given by the just the equation above.

- 'GraphSAGE_MEAN_MB_NS':

Same as 'GraphSAGE_MEAN_OTS', but using minibatches during training.

We recovered original results, as shown in Table 2.2.

2.5.3 Reproduction of Undirected Link Prediction Results

In the following, we will showcase our reproduction of some well known results in the literature that concern *undirected* link prediction i.e. link prediction performed on undirected graphs.

OGBL-DDI

This task has been taken from OGB. The ogbl-ddi dataset is an unweighted, undirected graph representing drug-drug interactions. Two drugs are connected if their joint usage leads to effects vastly different from the superposition of the effects of the two individual drugs. The task is to predict new drug-drug interactions using *Hits@K* metric with K=20. The split is a *protein-target* split, meaning that the training, validation and test edges are those contained in three different node-induced subgraphs. These node-induced subgraphs are constructed by splitting the drugs according to what protein do they target in the body. Therefore, we expect feature distributions to be non stationary among the three sets.

We followed this implementation. Therefore, the model is a GAE (see Section 2.2.4) that has a 'GraphSAGE_MEAN' encoder with $L = 2$ layers and a 'ReLU' activation after the first. The decoder is a 2-layer MLP which acts

	GAE	VGAE
AUC	0.92 ± 0.01	0.91 ± 0.006
F1	0.81 ± 0.01	0.799 ± 0.007
hitsk	0.66 ± 0.07	0.6 ± 0.05
AP	0.92 ± 0.02	0.905 ± 0.008
MRR	0.48 ± 0.06	0.42 ± 0.02

Table 2.3: Results of (V)GAE models over Cora dataset.

on the dot product of the nodes' embeddings and outputs a single logit. That means that the model predicts:

$$\hat{p}(e_{ij}) = \sigma(MLP(\vec{h}_i^{(L)} \cdot \vec{h}_j^{(L)}))$$

where σ is a sigmoid function.The remainder of model (hyper)parameters and training algorithm are as reported in the implementation. We reproduced the original results (0.54 Hits@K).

Reproduction of (V)GAE paper

We reproduced all results in [12]. We implemented the GAE and VGAE models as described in Section 2.2.4 and Section 2.2.5, and applied an *Effective* split on the *Cora* dataset as executed in the paper's repository.

We reproduced their results in Table 2.3.

Reproduction of (V)GNAE paper

The authors of [48] build on top of [12] by arguing that the encoder as implemented in the latter article leads to isolated nodes being all embedded near the origin, in order to prevent them from having high similarity scores between themselves or other nodes according to the usual dot product-based decoders. But nodes that are isolated at training time may actually develop new connections at test-time, and these would hardly be predictable by a model that embeds isolated nodes close to the origin.

As a solution, the authors propose to L2-normalize embeddings produced in the final layer of the encoder. Given the final embedding $\vec{h}_v^{(K)}$ of node v, the authors compute:

	GNAE	VNGAE
AUC	0.930 ± 0.004	~0.9
F1	0.793 ± 0.003	~0.76
AP	0.940 ± 0.003	~0.92

Table 2.4: Results of (V)GNAE models over Cora dataset.

$$\vec{n}_v^{(K)} = s\frac{\vec{h}_v^{(K)}}{||\vec{h}_v^{(K)}||_2}$$

Considering a GCN aggregation strategy (see Section 2.1.2), it therefore becomes:

$$\vec{h}_v^{(k)} = \frac{1}{d_v+1}\vec{n}_v^{(k-1)} + \sum_{u\in N(v)} \frac{1}{\sqrt{d_v+1}\sqrt{d_u+1}}\vec{n}_u^{(k-1)}$$

They name this aggregation strategy *Graph Normalized Convolutional Network* (*GNCN*).

The authors propose the *Graph Normalized Autoencoder* (*GNAE*) which is GAE with a 1-layer deep GNCN encoder and a dot-product decoder $\hat{P} = \sigma(Z^T Z)$, and the *Variational Graph Normalized Autoencoder* (*VGNAE*), which computes the mean using a GNCN and the standard deviation using a standard GCN.

We managed to reproduce their results using their 80% split on the Cora dataset (see Table 2.4)

2.5.4 Reproduction of Directed Link Prediction Results

In the following, we will showcase our reproduction of some well known results in the literature that concern *directed* link prediction i.e. link prediction performed on directed graphs.

Reproduction of Gravity-(V)GAE paper

The only decoder we used so far, namely

$$\hat{p}(e_{uv}) = \sigma(\vec{h}_v^{(L)} \cdot \vec{h}_u^{(L)})$$

will predict the same probability for e_{uv} and e_{vu}, thus making it unsuited for predicting directed links, where one direction may exist while the other doesn't. In [30], the authors propose a few decoders for directed link prediction. The first is the so-called *Gravity-Inspired* decoder, and it stems from considering nodes as "planets" and noting that while the gravitational force between two planets is the same, the acceleration each of them feels is different. In fact, given two planets of mass m_i and m_j separated by a distance r_{ij}, the modulus of the gravitational force between them will be:

$$F_{ij} = G\frac{m_i m_j}{r_{ij}^2}$$

but the acceleration that i feels is:

$$a_i = \frac{F_{ij}}{m_i} = \frac{Gm_j}{r_{ij}^2}$$

For computational reasons that will be more apparent later, we may take the logarithm of the latter equation:

$$\ln(a_i) = \ln(Gm_j) - \ln(r_{ij}^2) = \tilde{m}_j - \ln(r_{ij}^2)$$

Where we set $\tilde{m}_j := \ln(Gm_j)$.

The authors proceed by noting an asymmetric decoder can be constructed from this expression. Indeed, given node i's embedding $\vec{h}_i^{(L)}$, we may interpret its last dimension $\vec{h}_i^{(L)}[-1]$ as $\tilde{m}_i$, and the remainder of its elements as i's "coordinates" $\vec{h}_i^{(L)}[:-1] = \vec{r}_i$ so that $r_{ij}^2 = ||\vec{r}_i - \vec{r}_j||_2^2$.

Therefore, we may finally interpret $\ln(a_i)$ as the predicted unnormalized probability $\hat{\hat{p}}(e_{ij})$ that there exists a directed link from node i to node j:

$$\hat{\hat{p}}(e_{ij}) = \hat{m}_j - \ln(||\vec{r}_i - \vec{r}_j||_2^2)$$

Notice that the term $\ln(||\vec{r}_i - \vec{r}_j||_2^2)$ is symmetric in i and j, therefore it will be concerned with the existence of a link between i and j regardless of its direction. Only a single parameter $\hat{m}_j = \vec{h}_j^{(L)}[:-1]$ (that, note, depends on the target node only) decides directionality.

We may also add a learnable λ parameter that weighs the directed versus the undirected component of the decoder:

$$\hat{\hat{p}}(e_{ij}) = \hat{m}_j - \lambda \ln(||\vec{r}_i - \vec{r}_j||_2^2)$$

The authors also introduce another directed decoder, named *Source/Target* decoder. It considers the two halves of a node's embedding $\vec{h}_v^{(L)} \in \mathbb{R}^{d_L}$, namely:

$$\begin{cases} \vec{s}_v &= \vec{h}_v^{(L)}[:\frac{d_L}{2}] \\ \vec{t}_v &= \vec{h}_v^{(L)}[\frac{d_L}{2}:] \end{cases} \tag{2.21}$$

And the predicted probability for an edge from u to v is:

$$\hat{p}(e_{uv}) = \sigma(\vec{s}_u \cdot \vec{t}_v)$$

That is, $\vec{s}_v$ is the embedding of v when it acts as a *source*, while $\vec{t}_v$ when it acts as target.

Crucially, the authors note that random splits like those described in Section 2.4.2 would not suffice to properly train and test a directed link prediction model. In fact, a test set made of a random split of positive edges and an uncorrelated random sample of test edges would not properly assess the capability of the model to distinguish different orientations of an edge. Indeed, an undirected link prediction model would perform well on such dataset (more details in Chapter 3). Therefore, the authors devise three test-sets, each of them testing one aspect of directed link prediction:

1. *General* test set: like in Section 2.4.2, it is composed of a random split of positive edges and a random, uncorrelated sample of the same amount of negative edges;
2. *Biased* test set: it is composed of a random split of positive *unidirectional* edges and *their reverses as negatives*;

3. *Bidirectional* test set: it is composed of a random split of one direction of *bidirectional* positive edges and the reverses of positive unidirectional training edges.

In the above, a *unidirectional* positive/negative edge is a directed edge whose reverse is negative/positive, while a *bidirectional* (or *reciprocal*) positive/negative edge is a directed edge whose reverse is also positive/negative.

Clearly, the *Biased* test set evaluates the model's ability to distinguish edges' directions, while the *Bidirectional* test set is focused on assessing the model's capability of detecting bidirectionality.

Nevertheless, the authors do not try to learn a single model capable of tackling all three tasks simultaneously. Rather, they devise specific train/validation/test split for each case. In particular:

1. *General* train set: it is composed of all the positive edges not included in the *General* test set, and all possible negatives, just like in Section 2.4.2;
2. *Biased* train set: it is composed of all the positive edges not included in the *Biased* test set minus all bidirectional positives. All possible negatives are also added;
3. *Bidirectional* test set: it is composed of all the positive edges not included in the *Bidirectional* test set, and all possible remaining negatives.

The *General* task also has a validation set used to find the optimal λ, learning rate, number of epochs, etc. The same values are then brought over, dataset-wise, to the other tasks. Test metrics used are the ROC-AUC and AP-AUC (see Section 2.5.1).

As the following tables show, we managed to reproduce all results. Table 2.5 reports the results on the General task, Table 2.6 reports the results on the Biased task, Table 2.7 reports the results on the Bidirectional task.

Please compare our results with the original paper's in Table 2.8.

We also reproduced results using the SourceTarget GAE. Performances over the General task can be found in Table 2.9. Results on the Biased task can be found in Table 2.10, while Bidirectionality prediction results can be found in Table 2.11. Those are to be compared with the original paper's results in Table 2.12.

In all cases, the *altered* columns refer to results obtained by altering the

	Gravity-GAE	Gravity-VGAE
AUC	0.857 ± 0.002	0.9041 ± 0.0005
F1	0.74 ± 0.01	0.741 ± 0.007
hitsk	0.72 ± 0.01	0.754 ± 0.004
AP	0.902 ± 0.002	0.9279 ± 0.0007

Table 2.5: Results Gravity-(V)GAE models over the General task.

	Gravity-GAE	Gravity-VGAE	Gravity-GAE_altered	Gravity-VGAE_altered
AUC	0.83 ± 0.01	0.822 ± 0.003	0.848 ± 0.001	0.843 ± 0.001
F1	0.722 ± 0.003	0.742 ± 0.002	0.751 ± 0.006	0.771 ± 0.004
hitsk	0.445 ± 0.005	0.44 ± 0.01	0.37 ± 0.02	0.424 ± 0.003
AP	0.832 ± 0.005	0.82 ± 0.002	0.833 ± 0.003	0.834 ± 0.001

Table 2.6: Results of Gravity-(V)GAE models over the Biased task.

	Gravity-GAE	Gravity-VGAE	Gravity-GAE_altered	Gravity-VGAE_altered
AUC	0.775 ± 0.003	0.753 ± 0.01	0.794 ± 0.002	0.774 ± 0.003
F1	0.755 ± 0.006	0.74 ± 0.01	0.32 ± 0.01	0.4 ± 0.01
hitsk	0.55 ± 0.01	0.52 ± 0.04	0.57 ± 0.007	0.57 ± 0.02
AP	0.744 ± 0.003	0.7 ± 0.01	0.778 ± 0.004	0.742 ± 0.005

Table 2.7: Results Gravity-(V)GAE models over the Bidirectional task.

Model	General		Directional		Bidirectional	
	ROC-AUC	AP-AUC	ROC-AUC	AP-AUC	ROC-AUC	AP-AUC
G-GAE	87.79 ± 1.07	90.78 ± 0.82	83.18 ± 1.12	84.09 ± 1.16	75.57 ± 1.90	73.40 ± 2.53
G-VGAE	91.92 ± 0.75	92.46 ± 0.64	83.33 ± 1.11	84.50 ± 1.24	75.00 ± 2.10	73.87 ± 2.82

Table 2.8: Original G-(V)GAE performance on Cora dataset. *Courtesy of [12].*

	SourceTarget-GAE
AUC	0.86 ± 0.02
F1	0.5 ± 0.08
hitsk	0.66 ± 0.03
AP	0.894 ± 0.008

Table 2.9: Results of the SourceTarget-GAE model over the General task.

	SourceTarget-GAE	SourceTarget-GAE_altered
AUC	0.61 ± 0.01	0.878 ± 0.006
F1	0.39 ± 0.09	0.8 ± 0.008
hitsk	0.17 ± 0.02	0.56 ± 0.02
AP	0.63 ± 0.01	0.869 ± 0.009
AUC	0.61 ± 0.01	0.878 ± 0.006
F1	0.39 ± 0.09	0.8 ± 0.008
hitsk	0.17 ± 0.02	0.56 ± 0.02
AP	0.63 ± 0.01	0.869 ± 0.009

Table 2.10: Results of the SourceTarget-GAE model over the Biased task.

	SourceTarget-GAE
AUC	0.74 ± 0.02
F1	0.69 ± 0.01
hitsk	0.52 ± 0.07
AP	0.72 ± 0.03

Table 2.11: Results of the SourceTarget-GAE model over the Bidirectional task.

Model	General		Directional		Bidirectional	
	ROC-AUC	AP-AUC	ROC-AUC	AP-AUC	ROC-AUC	AP-AUC
ST-GAE	82.67 ± 1.42	83.25 ± 1.51	57.81 ± 2.64	57.66 ± 3.35	65.83 ± 3.87	63.15 ± 4.58

Table 2.12: Original Source/Target GAE results on Cora.. *Courtesy of [12].*

	Gravity-HGAE
AUC	0.864 ± 0.004
F1	0.755 ± 0.003
hitsk	0.726 ± 0.003
AP	0.906 ± 0.002

Table 2.13: Results of the Gravity-HGAE model over the General task.

Biased training set so that it contains its split of unidirectional positives and their reverses as negatives. This allows the training set to be statistically more similar to the test set, and therefore we observe a significative bump in performance, especially for the *SourceTarget* model. A Pytorch-Geometric based implementation of this work has been developed by me, reviewed by the lead author of [12] and published on GitHub.

We also attempted the general model using a *bidirectional* encoder:

$$\vec{h}_v^{(l+1)} = \sigma\left(W_{orig}^{(l+1)} \sum_{u \in N_{orig}(v)} \frac{\vec{h}_u^{(l)}}{|N_{orig}(v)|} + W_{rev}^{(l+1)} \sum_{u \in N_{rev}(v)} \frac{\vec{h}_u^{(l)}}{|N_{rev}(v)|}\right)$$

Where $N_{orig}(v)$ is the standard in-neighborhood of v, while $N_{rev}(v)$ is the out-neighborhood of v. Obtaining the results in Table 2.13(on general only).

Very much comparable with standard Gravity-GAE.

For our take on the prosecution of this work, please refer to Chapter 3.

Reproduction of AMLworld paper

We also performed a directed link prediction pass on a temporal split of the generated transaction network dataset found in [64].

The authors of [64] devise an agent-based model initialized by parameter distributions (avoiding the need of possessing real-world transaction data), which outputs a list of timestamped transactions between accounts. This work is the last in a relatively scarce literature which tries to generate transaction networks using ABMs (see [27] for an entry point to previous efforts).

The authors call their ABM *AMLworld*, and it models an entire financial system, with good actors (individuals, companies, banks,..) and bad actors

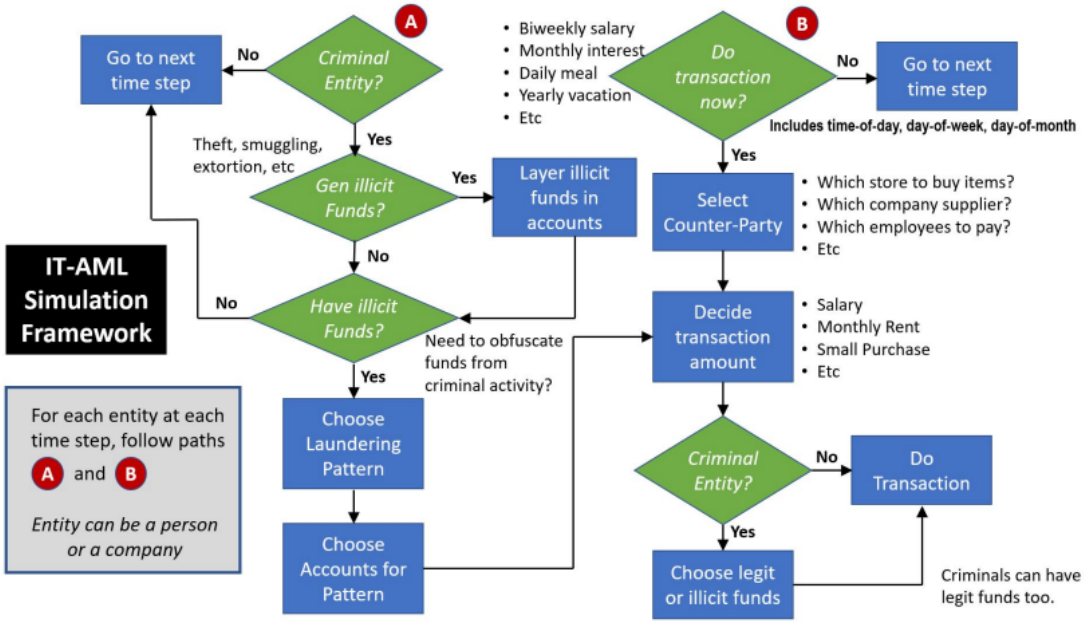


Figure 2.5: Schematic representation of the AMLworld ABM. The Layering process is detailed under A, while Integration under B. *Courtesy of [64]*

(launderers). Lawful transactions (such as salaries paid by companies to their employees) between legit accounts are allowed and actually constitute the majority. Companies may own other companies. For the criminal minority, the whole laundering process is modelled, from *Placement* through *Layering* to *Integration*. *Placement* refers to the way illicit funds are obtained. *Layering* constitutes the process of mixing the illict funds to the financial system. *Integration* means how the illict funds are spent. The ABM is made of both good and bad actors. Good actors always perform lawful transactions, while bad actors sometimes decide to do laundering. Placement may come from 9 different sources, layering (under A in the figure below) is performed by randomly choosing a pattern and the accounts to involve, while Integration (under B in the figure below) can be performed with a good actor counterpart. There are also many different types of lawful transactions. Please refer to Figure 2.5 for an illustration of how *Layering* and *Placement* are modelled.

The model is initialized via parameters distributions rather than real-world data. After it has run, it will produce two datasets. The first is a line-list of all transactions that occurred (see Figure 2.6).

The final ‘Is Laundering‘ column is binary 1-0 indicator. The second dataset

	Timestamp	From Bank	Account	To Bank	Account.1	Amount Received	Receiving Currency	Amount Paid	Payment Currency	Payment Format	Is Laundering
0	2022/09/01 00:08	11	8000ECA90	11	8000ECA90	3.195403e+06	US Dollar	3.195403e+06	US Dollar	Reinvestment	0
1	2022/09/01 00:21	3402	80021DAD0	3402	80021DAD0	1.858960e+03	US Dollar	1.858960e+03	US Dollar	Reinvestment	0
2	2022/09/01 00:00	11	8000ECA90	1120	8006AA910	5.925710e+05	US Dollar	5.925710e+05	US Dollar	Cheque	0
3	2022/09/01 00:16	3814	8006AD080	3814	8006AD080	1.232000e+01	US Dollar	1.232000e+01	US Dollar	Reinvestment	0
4	2022/09/01 00:00	20	8006AD530	20	8006AD530	2.941560e+03	US Dollar	2.941560e+03	US Dollar	Reinvestment	0
...	...	...	...	...	...	...	...	...	...	...	...
6924044	2022/09/10 23:39	71696	81B2518F1	71528	81C0482E1	3.346900e-02	Bitcoin	3.346900e-02	Bitcoin	Bitcoin	0
6924045	2022/09/10 23:48	271241	81B567481	173457	81C0DA751	1.313000e-03	Bitcoin	1.313000e-03	Bitcoin	Bitcoin	0
6924046	2022/09/10 23:50	271241	81B567481	173457	81C0DA751	1.305800e-02	Bitcoin	1.305800e-02	Bitcoin	Bitcoin	0
6924047	2022/09/10 23:57	170558	81A2206B1	275798	81C1D5CA1	4.145370e-01	Bitcoin	4.145370e-01	Bitcoin	Bitcoin	0
6924048	2022/09/10 23:31	170558	81A2206B1	275798	81C1D5CA1	3.427700e-02	Bitcoin	3.427700e-02	Bitcoin	Bitcoin	0

[6924049 rows x 11 columns]

Figure 2.6: The transaction line-list dataset.

```
BEGIN LAUNDERING ATTEMPT - FAN-IN:  Max 3-degree Fan-In
2022/09/01 02:38,001812,80279F810,0110,8000A94C0,10154.74,Australian Dollar,10154.74,Australian Dollar,ACH,1
2022/09/02 14:36,022595,80279F8B0,0110,8000A94C0,5326.79,Australian Dollar,5326.79,Australian Dollar,ACH,1
2022/09/03 14:09,001120,800E36A50,0110,8000A94C0,4634.81,Australian Dollar,4634.81,Australian Dollar,ACH,1
END LAUNDERING ATTEMPT - FAN-IN

BEGIN LAUNDERING ATTEMPT - FAN-IN:  Max 8-degree Fan-In
2022/09/01 03:17,003671,801BF8E70,002557,8016B3750,8099.96,Euro,8099.96,Euro,ACH,1
2022/09/01 06:27,015,80074C7E0,002557,8016B3750,10468.56,Euro,10468.56,Euro,ACH,1
2022/09/01 10:04,002557,80107C9A0,002557,8016B3750,10270.07,Euro,10270.07,Euro,ACH,1
2022/09/02 06:35,012,800A9B180,002557,8016B3750,15645.21,Euro,15645.21,Euro,ACH,1
2022/09/03 09:12,021393,801271170,002557,8016B3750,14139.75,Euro,14139.75,Euro,ACH,1
2022/09/03 13:45,002175,801E25F20,002557,8016B3750,6276.26,Euro,6276.26,Euro,ACH,1
2022/09/03 16:17,020,800043BE0,002557,8016B3750,1042.63,Euro,1042.63,Euro,ACH,1
2022/09/03 23:09,022124,8011D0180,002557,8016B3750,12795.57,Euro,12795.57,Euro,ACH,1
END LAUNDERING ATTEMPT - FAN-IN
```

Figure 2.7: A snippet of the dataset reporting generated laundering patterns

Table 4: Public Synthetic Data Statistics. **HI** = Higher Illicit (more laundering). **LI** = Lower Illicit.

	Small		Medium		Large	
Statistic	**HI**	**LI**	**HI**	**LI**	**HI**	**LI**
# of Days Spanned	10	10	16	16	97	97
# of Bank Accounts	515K	705K	2077K	2028K	2116K	2064K
# of Transactions	5M	7M	32M	31M	180M	176M
# of Laundering Transactions	3.6K	4.0K	35K	16K	223K	100K
Laundering Rate (1 per N Trans)	981	1942	905	1948	807	1750

Figure 2.8: Statistics of the synthetic datasets made available by [64]. *Courtesy of [64]*

leverages the white-box ABM dynamics to groups all illicit transactions into the patterns they belong to, by also stating the pattern type (see Figure 2.7).

The authors have made available online (link) the results of 6 runs. In short, two versions (HI, LI for Higher Illicit Rate, Lower Illicit Rate) of three datasets (Small, Medium, Large, that vary in simulation's length) have been produced, with the characteristics in Figure 2.8.

Figure 2.9 shows how AMLWorld compares to its contenders.

The authors proceed to perform laundering/non-laundering edge classification using GNNs and other algorithms with a 60-20-20 temporal split and topological features. Results are in Figure 2.10.

We performed a directed link prediction pass over the L1 dataset, mimicking

Table 1: Comparison to key previous synthetic AML data.

	MLDP	**AMLsim**	**AMLworld (This Work)**
Do placement?	✓		✓
Do layering?	✓	✓	✓
Do integration?	✓		✓
Model 8 key patterns?		✓	✓
Model other patterns?	✓		✓
Model multiple banks?			✓
Model multiple currencies?			✓
Complex entity graph?			✓
Model transfers, payments, credits, etc.?	Transfers only	Transfers only	✓
# of Transactions	2,340	1.32 M	180 M
Laundering rates	3/5	1/762	1/807 & 1/1,750

Figure 2.9: Comparison of AMLWorld with respect to main competitors. *Courtesy of [64]*

Table 2: Minority class F1 scores (%). HI indicates a higher illicit ratio. LI indicates a lower ratio.

Model	**HI**-*Small*	**LI**-*Small*	**HI**-*Medium*	**LI**-*Medium*	**HI**-*Large*	**LI**-*Large*
GIN [65, 25]	28.70 ± 1.13	7.90 ± 2.78	42.30 ± 0.44	3.86 ± 3.62	NA	NA
GIN + EU [11, 15]	47.73 ± 7.86	20.62 ± 2.41	49.26 ± 4.02	6.19 ± 8.32	NA	NA
PNA [60]	56.77 ± 2.41	16.45 ± 1.46	59.71 ± 1.91	27.73 ± 1.65	NA	NA
GFP [48, 49] + LightGBM [29]	62.86 ± 0.25	20.83 ± 1.50	59.48 ± 0.15	20.85 ± 0.38	48.67 ± 0.24	17.09 ± 0.46
GFP [48, 49] + XGBoost [16]	63.23 ± 0.17	27.30 ± 0.33	65.70 ± 0.26	28.16 ± 0.14	42.68 ± 12.93	24.23 ± 0.12

Figure 2.10: Minority class F1 scores (%). *Courtesy of [64]*

	Gravity-GAE
AUC	0.894 ± 0.003
F1	0.8399 ± 0.0009
AP	0.86 ± 0.01

Table 2.14: Results of the Gravity-GAE model over the General task on L1 dataset.

	Gravity-GAE
AUC	0.7827 ± 0.0002
F1	0.7416 ± 0.0005
AP	0.7252 ± 0.0002

Table 2.15: Results of the Gravity-GAE model over the Biased task on L1 dataset.

the *General* and *Biased* splits from [12]. For this task, we removed temporal information. The transformations we employed can be summarized as (they require familiarity with Pytorch Geometric):

1. Download data using Kaggle APIs;
2. Convert string Timestamps to UTC Epoch;
3. Sort Transactions by rows;
4. Convert hexadecimal Account IDs to integer;
5. Assign unique identifiers to banks, currencies, payment methods and related maps;
6. Construct Edge Index;
7. Construct torch_geometric.data.Data object and save.

Due to memory constraints, we weren't able to consider the whole adjacency matrix for training. Therefore, we employed Pytorch Geometric's 'RandomLinkSplit' with 'neg_sampling = 1' to obtain the *General*'s training set, while we opted for the *altered* strategy to get a training set for the *Biased* task. For the *General* setting, we obtained the results in Table 2.14. While in the *Biased* setting (altered strategy) we got the results in Table 2.15.

	Gravity-GAE
AUC	0.90908 ± 1e-05
F1	0.85816 ± 9e-05
AP	0.8657 ± 0.0005

Table 2.16: Results of the Gravity-GAE model over the General task on a temporal split of L1 dataset.

	Gravity-GAE
AUC	0.743 ± 0.002
F1	0.772 ± 0.001
AP	0.67 ± 0.002

Table 2.17: Results of the Gravity-GAE model over the Biased task on a temporal split of L1 dataset.

We also tried a temporal split, where, after removing duplicate edges, the remaining were split according to time in a 85-5-10 fashion. Note that this leads to a purely inductive setting (i.e. nodes in the test set have not been seen during training). We obtained in the *General* setting the results in Table 2.16, and in the *Biased* setting the results in Table 2.17.

The Bidirectional setting was not explored due to very low rate of bidirectional edges.

Chapter 3

Neural Directed Link Prediction is a Multiclass Edge Classification Task

I've already discussed Link Prediction, in both Undirected and Directed variants, in Section 2.4.2. We also hinted to the fact that there is an evident difference between Undirected and Directed Link Prediction (ULP and DLP respectively): while the former reduces to predicting the existence of an edge, the latter is also concerned with directionality and bi-directionality. As anticipated, we note an existing literature gap where most papers on Directed Link Prediction train and test their models only against existence, discarding the other two crucial aspects. In this chapter, we show that naively training a model for DLP via the usual train/validation/split approach with binary positive/negative labels leads to parameter configurations unsuited to discern directionality and bi-directionality, even after rebalancing the two classes. Furthermore, we propose to map DLP to a natural Multiclass Edge Classification task and demonstrate that this framework trains models simultaneously addressing all three aspects of DLP. We show the superiority of our approach in producing models capable of tackling the three subtasks DLP is composed of with respect to current standard methodologies.

3.1 Problem Formulation

As I've already noted in Section 2.4.2, within the context of Link Prediction, arcs that actually exist are usually referred to as *positive edges*, while non-existing links are named *negative edges*. Recent reviews on Link Prediction ([39], [71], [62]) mainly focus on the undirected variant with just one ([65]) also going into some depth regarding the directed case. Usually, models for Undirected Link Prediction (ULP models) are inductively biased towards predicting the same probability for an edge, irrespective of its orientation (which is often unassumed). The usual way of evaluating their generalization is by predicting the probabilities of (undirected) edges drawn from a random subset of all positive edges and a random selection of (undirected) negative edges from all possible negatives. Unfortunately, the same kind of split is often used to evaluate the performance of DLP models. We note, however, that by following this procedure an ULP model may seem to perform very well on DLP tasks (see Table 3.1). However, a ULP model would not be able to distinguish the two directions of an edge, and thus should not be deemed an effective model for DLP. To understand why an ULP model deceptively looks like performing so well against a DLP task, let us pose the following definitions.

Given a directed graph $G = (V, E)$, where $E = \{(u, v) | u, v \in V\}$, and given $u, v \in V$ we say that:

- (u, v) is *negative bidirectional* $\iff (u, v) \notin E \land (v, u) \notin E$;
- (u, v) is *negative unidirectional* $\iff (u, v) \notin E \land (v, u) \in E$;
- (u, v) is *positive unidirectional* $\iff (u, v) \in E \land (v, u) \notin E$;
- (u, v) is *positive bidirectional* $\iff (u, v) \in E \in (v, u) \notin E$;

Table 3.1: ROC-AUC, hits@20 and AP-AUC scored by a Graph Autoencoder from [12] trained and tested on a random split of Cora Dataset

Metric	Value
ROC AUC	0.829 ± 0.002
hits@20	0.59 ± 0.01
AP-AUC	0.872 ± 0.001

Due to networks being sparse, a test set made of a random sample of directed positives and a random sample of directed negatives, will most likely be constituted by positive unidirectional edges and negative bidirectional edges. Since pairs of reciprocal positive and negative edges are unlikely to be randomly sampled together, the ability of a model to properly distinguish the two directions of an edge is never really tested. Furthermore, a similar issue occurs for *bidirectionality*: an ULP model would always predict both positive and negative directed edges to be bidirectional, which is clearly not the case in directed networks.

Thus, a different evaluation setup is needed. The first work to become aware of these issues is [30], where the authors devise three test sets with different proportions of unidirectional and bidirectional edges:

- *General* test set: it is made of a random sample of positive edges and a random sample of negative edges
- *Directional* test set: it is made of a random sample of unidirectional positive edges and their reverses as negatives
- *Bidirectional* test set: it is made of a random sample of bidirectional positive edges and a random sample of unidirectional negative edges

NB: In the original paper, the *Directional* test set is called *Biased*, but we changed the name here for consistency.

The *General* test set evaluates the prediction of the existence of directed edges, without much examining the ability of the model to establish direction. the *Directional* test set assesses the ability of the model to predict the direction of an edge, conditioned on the existence of one direction or the other. The *Bidirectional* test set evaluates the capability of a model to predict bidirectionality, once again conditioned on the existence of at least one of the two directions. If we were to evaluate a ULP model over the Directional or the Bidirectional tasks, it would obviously perform poorly. Surprisingly, the only paper we are aware of that tests DLP models against all three subtasks is [30]. [52] performs the *General* and *Directional* tests only. Even so, both [30] and [52] fail to find a single model that performs well on all three aspects of DLP. In fact, [30] also devises different training sets for each task, while [52] employs different hyperparameter values. In this chapter, we will propose a simple framework to train DLP models that allows them to perform well on all three tasks separately. Following [12] and [30] we note that during training, one would like to leverage information from all edges,

including all possible negatives. But this produces a training set with few, sparse unidirectional positives and orders of magnitudes more bidirectional negatives. Such configuration may lead the model, even after rebalancing the weights of positives and negatives inside the loss, to give little importance to directionality: after all, mistaking all the few negatives unidirectional for positives would only slightly increase the loss, while there is a huge incentive to predict all negatives as bidirectional. The natural solution to this issue is to simultaneously rebalance, within the loss, the contributions of all four categories of edges: positive unidirectional, positive bidirectional, negative unidirectional and negative bidirectional. This entails mapping the DLP task to a 4-class edge classification task. In the remainder, we prove that DLP models trained following this approach perform equally well on all three subtasks, while DLP models trained as if DLP was a binary classification task on a naive split tend to perform well only with respect to the *General* task. We believe our framework is the correct, principled approach to DLP, and is potentially applicable to any DLP model by adopting the slight modifications we will describe below.

This chapter is organized as follows: Section 3.2 reviews related works on DLP; Section 3.3 describes the details of our framework; Section 3.4 breaks down the training/validation/test setups that we use and presents results; Section 3.5 concludes the chapter.

3.2 Related Works on Directed Link Prediction

As noted before, three out of the the four most recent reviews on Link Prediction ([39], [71], [62]) mainly focus on the undirected variant with just one [65] also considering the directed case.

Non-neural model for DLP exist, for example [21], where the authors employ machine learning classifiers over extensions to 3-motifs of common similarity measures.

Among the neural models, GNN-based autoencoders represent the main approach. [12] is the seminal work in this regard, where GAE and VGAE were first proposed for undirected graphs. The training set is given by a random

split of all positive edges and all remaining possible negatives (including positive test and validation edges); positive and negative contributions are balanced in the training loss. [48] proposes the (Variational) Normalized Graph Autoencoder, where the norm of the latent dimensions is normalized to unity in order to prevent isolated nodes from collapsing near the origin. [30] proposes the Gravity (Variational) Graph Autoencoder and the Source/Target (Variational) Graph Autoencoder. The first model relies on one single latent space dimension to encode directionality, while the latter virtually learns two separate embeddings for each node, one for when it asct as a source and the other for when it acts as a target. This paper, as mentioned earlier, also proposes the three General, Directional and Bidirectional tasks. For every task, a suitable training set (and hyperparameter values) is devised, resulting in one different model per task. The implementation builds on that of [12]. [58] extends the Weisfeiler-Leman algorithm to the directed case, and proposes an MPNN based on it. Its implementation builds on that of [30]. The model is trained and tested only on the General task. [46] devises a GCN based on a laplacian for directed graphs obtained by normalizing a degree-weighted random walk transition matrix using PageRank's limit distribution. This laplacian is used in an inception block to define the first-order aggregation strategy. The model is tested using random train/val/test split. [52] proposes another laplacian for directed graphs, and defines a GCN based on it. Interestingly, it uses a training setup with three out of the aforementioned four edge classes, but then it is not applied to all three tasks and for each task, a different hyperparameter set is used. We will show that using our framework, a good compromise of the performance across all three DLP subtasks is obtained. Our framework could be readily retrofitted to the aforementioned models to increase their overall DLP performance.

3.3 Methods

We argue that the correct training set for a DLP task should be composed of a random split of all positive edges and, given this split, all possible negative edges. Such training set retains in fact maximum information, as noted in [12], [30], [58]. After balancing the contributions of positive and negative edges, training on this set produces models that generally perform well on the General sub-task, but suboptimally on the other tasks (see even rows of Section 3.4, Section 3.4, Section 3.4 and Section 3.4). As noted before, we hypothesize that this is due to an imbalance between unidirectional and

bidirectional edges. Therefore, we aim to balance them in the loss without sacrificing the balancing between positives and negatives. This implies mapping DLP to a 4-class classification task, where the classes are:

1. negative bidirectional (nb)
2. negative unidirectional (nu)
3. positive unidirectional (pu)
4. positive bidirectional (pb)

So that, in the loss, we may weigh them appropriately. In the following, we will name this 4-class classification framework for DLP Multiclass Framework for Directed Link Prediction, or *MFDLP*. Given a GNN model that computes d_K dimensional embeddings $\vec{z}_v^{(K)} \forall v \in V$, we may compute logits for each class by applying an MLP to the concatenation of the embeddings. The MLP must take $2d_K$ input dimensions and output 4 logits, and can be arbitrarily deep:

$$[\hat{l}_{uv}^{nb}, \hat{l}_{uv}^{nu}, \hat{l}_{uv}^{pu}, \hat{l}_{uv}^{pb}] = \text{MLP}(\vec{z}_u^{(K)} || \vec{z}_v^{(K)}) \tag{3.1}$$

And then using cross-entropy loss, which is essentially a SoftMax followed by a Negative LogLikelihood loss. Referring to an *Effective* split (Section 2.4.2) where E_{m+s} is the training set, the cross-entropy loss can be written as:

$$\mathcal{L}_T = -\sum_{e_{uv} \in E_{m+s}} w_{y_{uv}} \ln(\hat{p}(e_{uv})) \tag{3.2}$$

Where $\hat{p}(e_{uv})$ is the probability that the model predicts for edge e_{uv} and $w_{y_{uv}}$ (with $y_{uv} \in \{nb, nu, pu, pb\}$) is the weight associated to the class e_{uv} belongs to. $w_{y_{uv}}$ is computes before training as (assuming $x \in \{nb, nu, pu, pb\}$ is the class with the highest cardinality n_x):

$$w_{y_{uv}} = \frac{n_x}{n_{y_{uv}}}$$

In order to equally weight all four classes. The right-hand side of Equation (3.2) can be normalized by $\frac{1}{\sum_{e_{uv} \in E_{m+s}} w_{y_{uv}}}$ in order to obtain the mean cross-entropy.

MFDLP is also compatible with more sophisticated models such as [30] or [58] that make use of specific decoders which output only one logit $\hat{l}_{uv}$, namely the probability of an edge. We can turn the standard 1-class classification task for DLP into a 4-class classification task by transforming the logits into probabilities via e.g. a sigmoid:

$$\hat{p}_{uv} = \sigma(\hat{l}_{uv})$$

And then we define:

$$\begin{aligned}[\hat{l}^{nb}_{uv}, \hat{l}^{nu}_{uv}, \hat{l}^{pu}_{uv}, \hat{l}^{pb}_{uv}] = [&(1-\hat{p}_{uv})(1-\hat{p}_{vu}),\\ &(1-\hat{p}_{uv})\hat{p}_{vu},\\ &\hat{p}_{uv}(1-\hat{p}_{vu}),\\ &\hat{p}_{uv}\hat{p}_{vu}]\end{aligned}$$

Which would then be passed to a Negative Log Likelihood Loss.

As models, we used the Gravity-GAE (GGAE) and the Source/Target-GAE (STGAE) from [30], and we also devised a new decoder (MIX) which is a middle ground between the two. Similarly to Source/Target-GAE, MIX computes one embedding $\vec{z}_v$ for each node v, which is then split into three equal parts. The first part of the embedding is used for when v acts as a source ($\vec{s}_v$), the third part for when it acts as a target ($\vec{t}_v$), and the middle one ($\vec{t}_v$) represent the undirected part of the information. Using this concept, we employ two different decoders:

- Dot-Product based (MIX):

$$\hat{l}_{uv} = \vec{s}_u \cdot \vec{t}_v + \lambda(\vec{t}_u \cdot \vec{t}_v)$$

- Distance-Based (MIXDIST):

$$\hat{l}_{uv} = -\ln(||\vec{s}_u - \vec{t}_v||_2^2) - \lambda \ln(||\vec{t}_u - \vec{t}_v||_2^2)$$

Where the parameter λ weighs the relative importance of the two terms. This models represent a middle ground between the Gravity-GAE and the Source/Target-GAE in that, like Gravity-GAE, they allow for weighting the

directed and undirected parts of the information, but simultaneously grants more dimensions to work with for directionality (like the Source/Target-GAE does). The choice between a dot-product or a distance based approach is treated as hyperparameter tuning.

For the Dot-Product based, we also try another variant where final embeddings are l2-normalized before being passed to the decoder (MIXNORM), see [48].

Finally, we also try a 1-layer MLP decoder like that of Equation (3.1) (MLP).

3.4 Results

In the following, we report the performances of the three Graph Autoencoders described in section 3.3. The odd rows of Section 3.4, Section 3.4, Section 3.4 and Section 3.4 illustrate the performances of Gravity-GAE, SourceTarget-GAE and Mixed-GAE trained using *MFDLP* across the three tasks, while the even rows report the performances of the same models naively trained by just balancing positives and negatives. All datasets were split into train/val/test sets as follows. After removing all positive self-loops, 10% (5%) of all positive unidirectionals were used as positive edges of the *Directional* test set (validation set), and their reverses as negatives. Afterward, one direction of 30% (15%) of all bidirectional edges was used as positives for the *Bidirectional* test set. All other positive edges were used for training, together with all possible negative edges. The *General* test (validation) set was formed by using the positive edges of the *Directional* and *Bidirectional* test (validation) sets as positive edges, and sampling an equal amount of random negative edges. The training set was constituted by all remaining positive edges and all possible negatives, given the training positives. All models were trained once and then tested on the three sub-tasks. Performances over 5 random initializations were averaged. As validation loss, the sum of the ROC AUC and the AP AUC were used. The optimal number of epochs was detected using Early Stopping looking at a single validation set loss given by the sum of the three losses over the three validation sets. The other models' parameters were either taken from [30] or fitted to reproduce the results in that paper. We used Pytorch Geometric, and the reimplementation of [30]'s models in Pytorch Geometric was endorsed by the original author and can be found in this repository: [CODE WILL BE PUBLISHED WHEN PAPER IS ACCEPTED].

Model	General	Directional	Bidirectional
GAE	79 ± 2	50 ± 0	63 ± 3
G-GAE + MFDLP	81.5 ± 0.2	76.3 ± 0.2	81 ± 1
G-GAE	79.4 ± 0.7	55 ± 4	63 ± 6
ST-GAE + MFDLP	74 ± 4	76 ± 2	83 ± 3
ST-GAE	82.4 ± 0.7	57 ± 1	70 ± 5
MIX + MFDLP	78 ± 4	74 ± 2	84 ± 1
MIX	82.4 ± 0.7	57 ± 1	70 ± 5
MIXDIST + MFDLP	0.807 ± 0.008	0.744 ± 0.006	0.82 ± 0.01
MIXDIST	0.786 ± 0.008	0.502 ± 0.002	0.71 ± 0.03
MIXNORM + MFDLP	0.796 ± 0.005	0.775 ± 0.002	0.79 ± 0.005
MIXNORM	0.83 ± 0.01	0.539 ± 0.005	0.75 ± 0.02
MLP + MFDLP	0.623 ± 0.007	0.78 ± 0.01	0.81 ± 0.01
MLP	0.654 ± 0.004	0.838 ± 0.004	0.798 ± 0.008

Table 3.2: ROC-AUC test scores of various AE models on Cora Dataset, trained both naively (random train/val/test split and balancing positives and negatives) and with our framework. Scores are in %.

Model	General	Directional	Bidirectional
GAE	82 ± 2	50 ± 0	64 ± 5
G-GAE + MFDLP	84.8 ± 0.4	75.2 ± 0.2	82 ± 1
G-GAE	86.0 ± 0.5	55 ± 4	58 ± 7
ST-GAE + MFDLP	76 ± 5	77 ± 1	85 ± 2
ST-GAE	85 ± 2	62 ± 1	73 ± 4
MIX + MFDLP	77 ± 2	75 ± 2	84 ± 1
MIX	85 ± 2	62 ± 1	73 ± 4
MIXDIST + MFDLP	0.83 ± 0.01	0.757 ± 0.005	0.834 ± 0.008
MIXDIST	0.852 ± 0.006	0.505 ± 0.002	0.73 ± 0.03
MIXNORM + MFDLP	0.837 ± 0.006	0.783 ± 0.003	0.81 ± 0.005
MIXNORM	0.874 ± 0.008	0.57 ± 0.01	0.79 ± 0.02
MLP + MFDLP	0.635 ± 0.007	0.795 ± 0.008	0.83 ± 0.01
MLP	0.643 ± 0.003	0.831 ± 0.003	0.825 ± 0.007

Table 3.3: AP-AUC test scores of various AE models on Cora Dataset, trained both naively (random train/val/test split and balancing positives and negatives) and with our framework. Scores are in %.

Model	General	Directional	Bidirectional
GAE	0.75 ± 0.01	0.5 ± 0.0	0.47 ± 0.06
G-GAE + MFDLP	0.76 ± 0.005	0.756 ± 0.004	0.87 ± 0.02
G-GAE	0.714 ± 0.004	0.51 ± 0.02	0.64 ± 0.05
ST-GAE + MFDLP	0.66 ± 0.01	0.81 ± 0.01	0.8 ± 0.02
ST-GAE	0.771 ± 0.009	0.549 ± 0.006	0.63 ± 0.06
MIX + MFDLP	0.68 ± 0.01	0.78 ± 0.02	0.74 ± 0.07
MIX	0.78 ± 0.008	0.563 ± 0.003	0.69 ± 0.07
MIXDIST + MFDLP	0.732 ± 0.002	0.746 ± 0.009	0.7 ± 0.07
MIXDIST	0.71 ± 0.01	0.501 ± 0.001	0.63 ± 0.06
MIXNORM + MFDLP	0.719 ± 0.005	0.786 ± 0.007	0.823 ± 0.007
MIXNORM	0.74 ± 0.02	0.5496 ± 0.0007	0.83 ± 0.03
MLP + MFDLP	0.592 ± 0.008	0.76 ± 0.05	0.78 ± 0.04
MLP	0.66 ± 0.04	0.9 ± 0.1	0.87 ± 0.03

Table 3.4: ROC-AUC test scores of various AE models on Citeseer Dataset, trained both naively (random train/val/test split and balancing positives and negatives) and with our framework.

We also plotted the shifts in performance due to the usage of our framework versus performing binary classification on a naive Effective split in the three subtasks, one against the other. Therefore, Figure 3.1 plots the shift in ROCAUC performance in the Directional vs General tasks, Figure 3.2 in the Bidirectional vs General tasks and Figure 3.3 in the Bidirectional vs Directional tasks. What we observe is that the application of *MFDLP* benefits all distance-based models, while it manages to find a better compromise between the General vs Directional/Bidirectional tasks for all dot-product based models, and has little to negative effect on the MLP model. The exact reasons for these discrepancies is subject of current research.

3.5 Discussion

In this chapter, we have stressed the need to evaluate Directed Link Prediction using three non-overlapping test sets evaluating existence, directionality and bidirectionality retrieval respectively, heavily inspired by [30]. That is, DLP is made of three sub-tasks. We named the three corresponding test

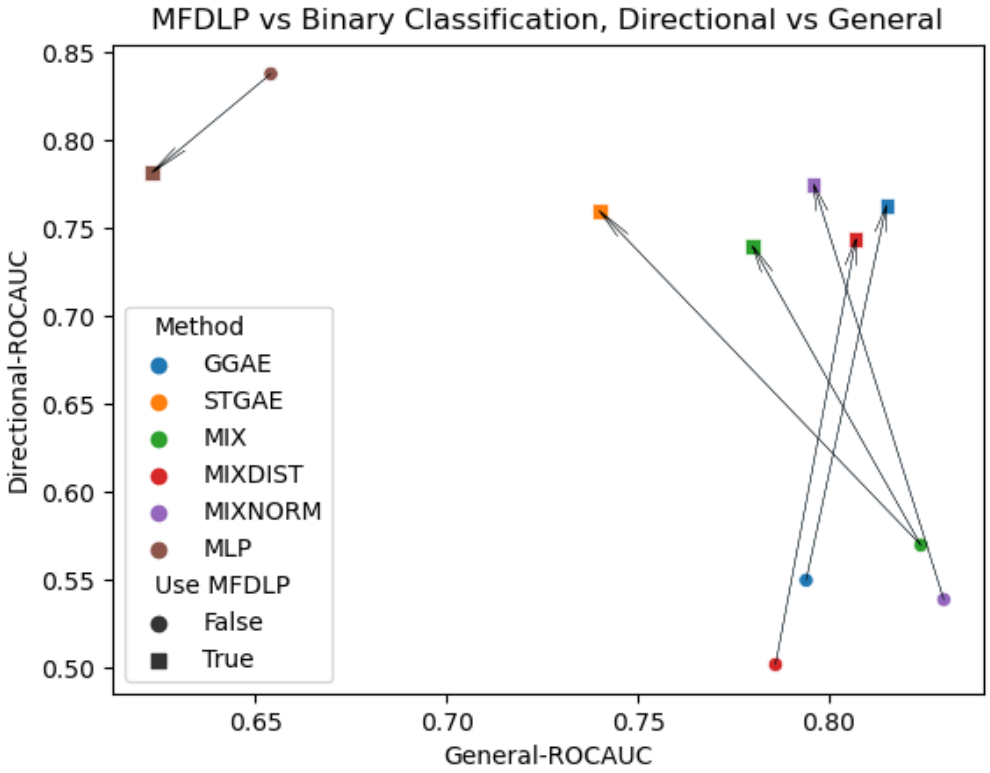


Figure 3.1: Shifts in ROCAUC performance due to the usage of our framework versus performing binary classification on a naive Effective split for the Cora dataset (Directional vs General). For each model, an arrow starts from the ROCAUC performances obtained by training via Binary Classification, and points to the ROCAUC performances obtained by using MFLDP.

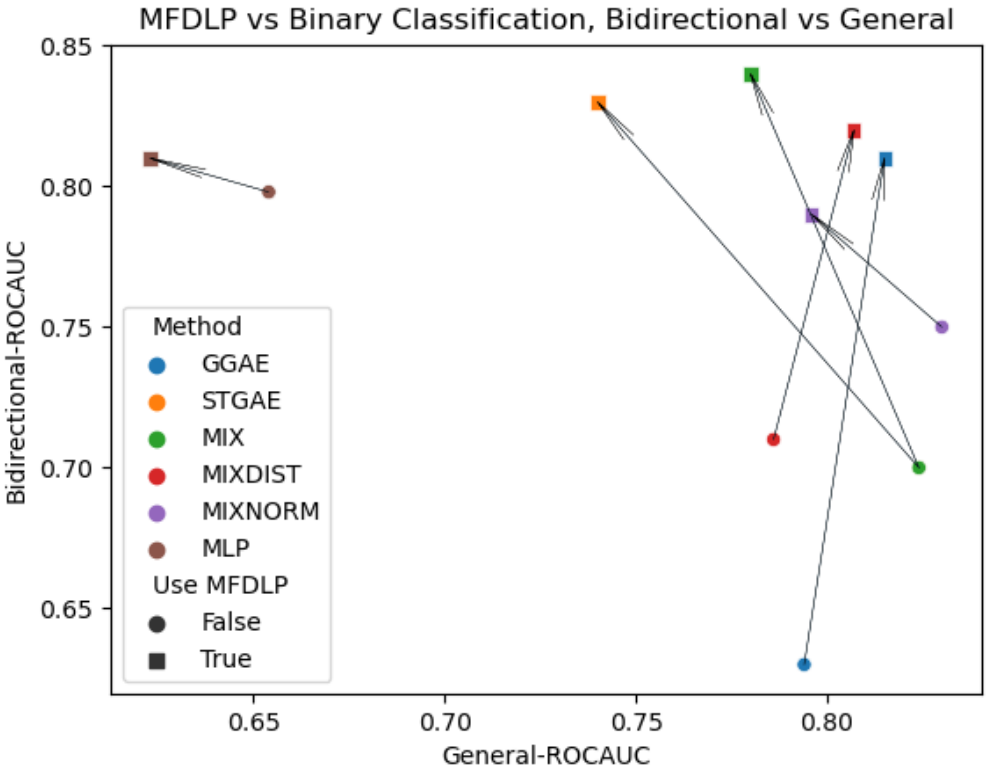


Figure 3.2: Shifts in ROCAUC performance due to the usage of our framework versus performing binary classification on a naive Effective split for the Cora dataset (Bidirectional vs General). For each model, an arrow starts from the ROCAUC performances obtained by training via Binary Classification, and points to the ROCAUC performances obtained by using MFLDP.

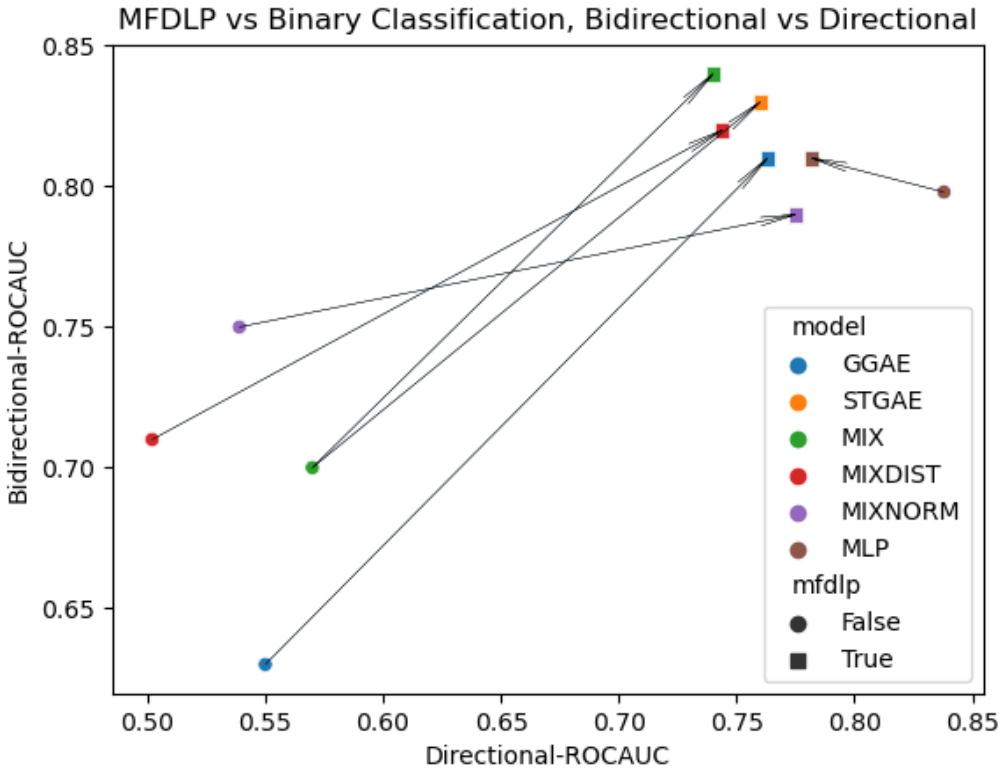


Figure 3.3: Shifts in ROCAUC performance due to the usage of our framework versus performing binary classification on a naive Effective split for the Cora dataset (Bidirectional vs Directional). For each model, an arrow starts from the ROCAUC performances obtained by training via Binary Classification, and points to the ROCAUC performances obtained by using MFLDP.

Model	General	Directional	Bidirectional
GAE	0.8 ± 0.01	0.5 ± 0.0	0.48 ± 0.03
G-GAE + MFDLP	0.795 ± 0.003	0.75 ± 0.002	0.87 ± 0.01
G-GAE	0.778 ± 0.009	0.52 ± 0.03	0.7 ± 0.04
ST-GAE + MFDLP	0.677 ± 0.008	0.841 ± 0.006	0.82 ± 0.02
ST-GAE	0.78 ± 0.01	0.604 ± 0.005	0.64 ± 0.08
MIX + MFDLP	0.7 ± 0.01	0.82 ± 0.02	0.78 ± 0.06
MIX	0.78 ± 0.003	0.613 ± 0.003	0.71 ± 0.05
MIXDIST + MFDLP	0.755 ± 0.002	0.767 ± 0.009	0.71 ± 0.07
MIXDIST	0.775 ± 0.005	0.503 ± 0.001	0.7 ± 0.07
MIXNORM + MFDLP	0.764 ± 0.004	0.813 ± 0.006	0.82 ± 0.008
MIXNORM	0.79 ± 0.01	0.606 ± 0.006	0.86 ± 0.02
MLP + MFDLP	0.62 ± 0.02	0.79 ± 0.07	0.79 ± 0.03
MLP	0.67 ± 0.02	0.87 ± 0.09	0.881 ± 0.002

Table 3.5: AP-AUC test scores of various AE models on Citeseer Dataset, trained both naively (random train/val/test split and balancing positives and negatives) and with our framework.

sets *General*, *Directional* and *Bidirectional* respectively. We found very few previous works that took this crucial nuance into account, let alone training models to perform well on all three sub-tasks simultaneously. Most other works focused on the existence retrieval, but we showed that due to how the *General* test set is devised, even undirected models perform well. Moreover, directed models naively trained, although outperforming undirected models, are generally not able to attain similar performance on the other two aspects of DLP. We hypothesized that this was due to a lack of insight where not only positive and negative edges need to be rebalanced during training, but also unidirectional and bidirectional edges. So we proposed *MFDLP*, a framework that maps DLP to a 4-class classification task taking into account all (positive, negative) and (unidirectional, bidirectional) combinations, enabling models to find a better compromise between the three aspect of DLP, and we compared a few models' performances when using and not using *MFDLP*.

Chapter 4

Temporal Graph Representation Learning

Temporal graphs [31] are the most natural way of representing many physical systems such as social networks. On the other hand, temporality is the second aspect of transaction networks (besides directionality) that we need our in-development representation learning framework to incorporate. Most of the recent literature [55], [67], [70], deal with temporality by adapting GNNs to temporal graphs, producing models known as *Temporal Graph Neural Networks* (*TGNNs*). We will follow the overview given by [70], since, to the best of our knowledge, it is the only one that attempts to deliver a conceptual taxonomy of TGNNs (Section 4.1). For every branch of the taxonomy, one representative model will be selected and explained in detail. During this phase, we contribute the mathematical details of Model Agnostic Meta Learning [14] (a technique instrumental to training a particular class of TGNNs), which are only summarily sketched in the original and related works. Finally, in Section 4.3 we will argue the disadvantages of these models and propose to tackle the issue of embedding temporality by first computing a time-preserving static representation of the dynamic graph, and later apply a standard GNN. In the same section, we demonstrate the superiority of this approach when compared to TGNNs in a dynamic node classification task from TGB.

4.1 Temporal Graph Neural Networks

Temporal graphs are graphs where *events* occur, where an event is a node/edge insertion/deletion at a time t. Moreover, node and edge features are allowed to change in a temporal graph.

Reference [70] defines a temporal graph as follows:

Definition 4.1.1 *A Temporal Graph is a tuple $G_T = (V, E, V_T, E_T)$ where V and E are, respectively, the set of all possible nodes and edges appearing in a graph at any time, while*

$$\begin{cases} V_T & := \{(v, x^v, t_s, t_e) | v \in V, x^v \in \mathbb{R}^{d_V}, t_s \leq t_e\} \\ E_T & := \{(e, x^e, t_s, t_e) | e \in E, x^e \in \mathbb{R}^{d_E}, t_s \leq t_e\} \end{cases} \tag{4.1}$$

are the temporal nodes and edges, with time dependent features and initial and final timestamps. A set of temporal graphs is denoted as $\mathcal{G}_T$.

The author continue by distinguishing between *Snapshot-Based Temporal Graphs* (*STGs*) and *Event-Based Temporal Graphs* (*ETGs*). Intuitively, STGs are temporal graphs constituted by a sequence of static graphs named snapshots, while ETGs allow for events (node/edge insertion/deletion) to happen continuously at any time.

STGs are concretely defined as:

Definition 4.1.2 *Let $t_1 < t_2 < ... < t_n$ be the ordered set of all timestamps t_s, t_e occurring in a TG G_T . Set:*

$$\begin{cases} V_i & := \{(v, x^v, t_s, t_e) | v \in V, x^v \in \mathbb{R}^{d_V}, t_s \leq t_i \leq t_e\} \\ E_i & := \{(e, x^e, t_s, t_e) | e \in E, x^e \in \mathbb{R}^{d_E}, t_s \leq t_i \leq t_e\} \end{cases} \tag{4.2}$$

and define the snapshots $G_i := (V_i, E_i), i = 1, ..., n$. Then a Snapshot-based Temporal Graph representation of G_T is the sequence:

$$G_T^S := \{(G_i, t_i) | i = 1, ..., n\}$$

of time-stamped static graphs.

While *ETGs* are given by a stream of events:

Definition 4.1.3 *Let G_T be a TG, and let ϵ denote one of the following events:*

- *Node insertion $\epsilon_V^+ := (v,t)$: the node v is added to G_T at time t, i.e., there exists $(v, x^v, t_s, t_e) \in V_T$ with $t_s = t$.*
- *Node deletion $\epsilon_V^- := (v,t)$: the node v is removed from G_T at time t, i.e., there exists $(v, x^v, t_s, t_e) \in V_T$ with $t_e = t$.*
- *Edge insertion $\epsilon_E^+ := (e,t)$: the edge e is added to G_T at time t, i.e., there exists $(e, x^e, t_s, t_e) \in E_T$ with $t_s = t$.*
- *Edge deletion $\epsilon_E^- := (e,t)$: the edge e is removed from G_T at time t, i.e., there exists $(e, x^e, t_s, t_e) \in E_T$ with $t_e = t$.*

An Event-based Temporal Graph representation of TG is a sequence of events:

$$G_T^E := \{\epsilon | \epsilon \in \{\epsilon_V^+, \epsilon_V^-, \epsilon_E^+, \epsilon_E^-\}\}$$

Here it is implicitly assumed that node and edge events are consistent (e.g., a node deletion event implies the existence of an edge deletion event for each incident edge).

Reference [70] asserts that TGNNs architectures can be distinguished in two groups: *Snapshot-based* architectures that operate on STGs, and *Event-based* architectures that can be trained over ETGs. Snapshot-based models are further distinguished in two classes: *Model Evolution methods* refer to those models that evolve their parameters from snapshot to snapshot (see Section 4.1.1), while *Embedding Evolution methods* evolve the embeddings only (see Section 4.1.2). Event-based models are also further distinguished in two categories: *Temporal Embedding methods* and *Temporal Neighborhood methods* (see Section 4.1.3 and Section 4.1.4 respectively). We present one model per each of the four classes below.

4.1.1 Model Evolution methods (EvolveGCN)

EvolveGCN ([29]) is a TGNN architecture designed for STGs. To give an initial, simplified idea, EvolveGCN is completely specified once a K-MPNN $\mathcal{M}_t^K = (M_t^{(k)})_{k=1}^K$ (a GCN in the original paper) is specified for each snapshot t and an RNN architecture (GRUs and LSTMs in the paper, see D2L book for

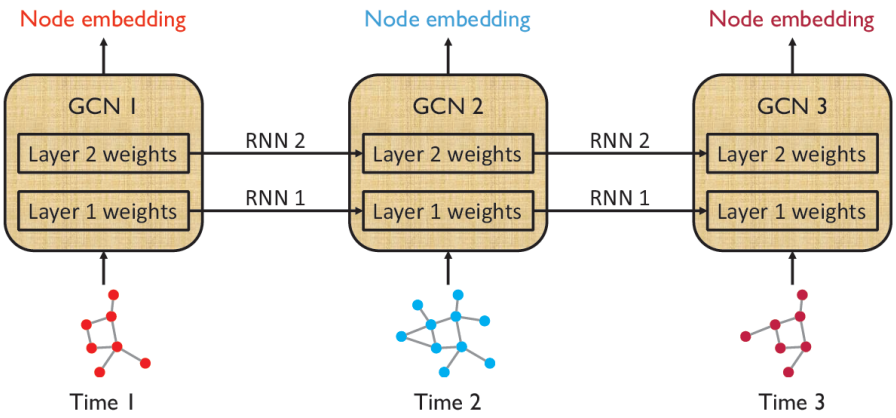


Figure 4.1: Node features are passed as input, while parameters are evolved using RNNs. *Courtesy of [29].*

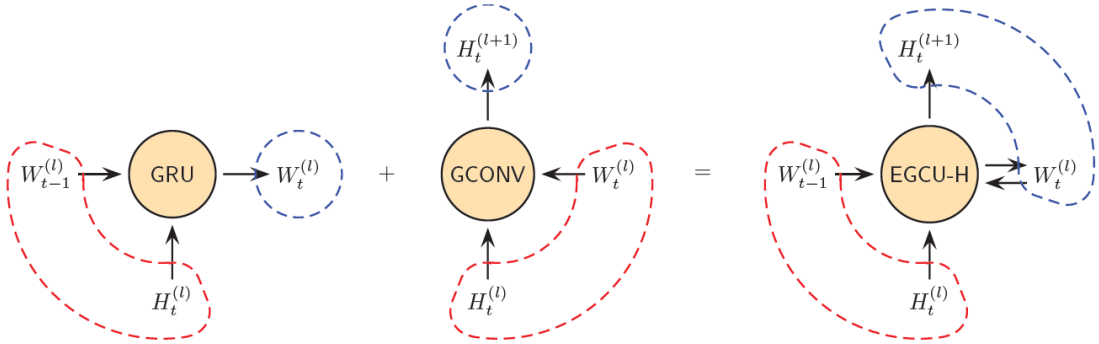
a guide on RNNS) is selected. At a high level, $\mathcal{M}_0^K$ is run over the first snapshot ($t = 0$) to compute the first embeddings $H_0^{(K)} = \mathcal{M}_0^K(H_0^{(0)})$. Then, the node features of the second snapshot $H_1^{(0)}$, together with the RNN-evolved parameters $W_1^{(1)} = RNN(H_1^{(0)}, W_0^{(1)})$ are used to compute the first-layer node embeddings for the second snapshot $H_1^{(1)} = \mathcal{M}_1^{(1)}(H_1^{(0)}; W_1^{(1)})$. Similarly for the other layers and time steps. Figure 4.1 illustrates the process.

The authors devise two versions of EvolveGCN, depending on the interpretation of the parameters $W_t^{(k)}$ with respect to the RNN. If the $W_t^{(k)}$ are treated as their internal state (and consequently the embeddings $H_t^{(k)}$ will be the input), then the model is named EvolveGCN-H, while if the parameters are the input of the RNN, then the model is referred to as EvolveGCN-O. EvolveGCN-H uses a GRU unit as RNN, while EvolveGCN-O employs an LSTM, delegating the role of the internal states to its memory. A pictorial representation of the two model may look like Figure 4.2.

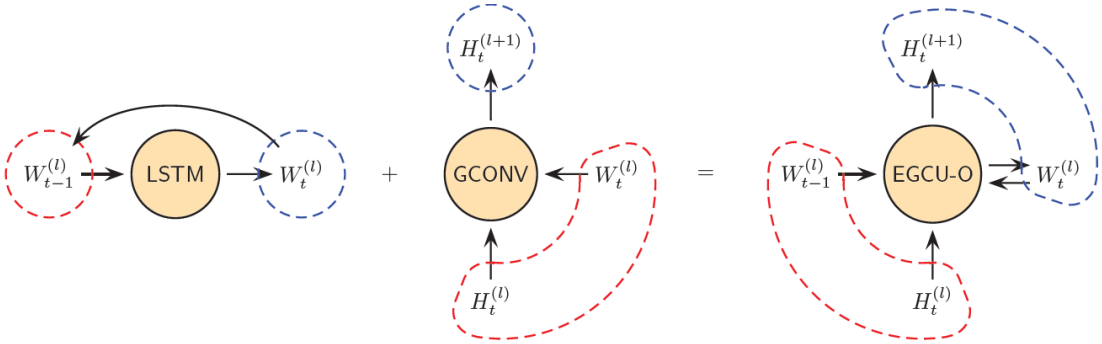
The authors later deal with technical issues such as a varying embedding size between layers, and adapting RNNs to matrix input/outputs rather than vectors.

4.1.2 Embedding Evolution methods (ROLAND)

Differently from EvolveGCN, ROLAND [63] directly predicts the evolution of node embeddings of STGs rather than GNN parameters. ROLAND allows



(a) EvolveGCN-H, where the GCN parameters are hidden states of a recurrent architecture that takes node embeddings as input.



(b) EvolveGCN-O, where the GCN parameters are input/outputs of a recurrent architecture.

Figure 4.2: EvolveGCN-H and EvolveGCN-O architecture. *Courtesy of [29].*

for lifting static GNNs to STGs, by combining them with RNNs. In particular, embeddings will be computed as $H_t^{(k)} = GRU(H_{t-1}^{(k)}, GNN(H_t^{(k-1)}))$. Due to this specific embedding evolution, ROLAND only needs to keep the last snapshot and embedding matrix to compute the next, allowing the approach to be agnostic to the number of snapshot and therefore scale better than previous endeavors. This mechanism will be called *live-update*, and will also allow for a more comprehensive evaluation setting. Moreover, to take seasonality effects into account, the model at time step t is not a fine tuning of the model trained at time $t-1$, rather, a meta-learning setting is introduced where a meta-model is responsible for initializing the model's parameters at each time step t, after which the fine-tuning begins. We will now explore each component of ROLAND.

Hierarchical Node Update

The time-dependent component of node embeddings' dynamics is taken care of by introducing a further 'UPDATE' step after the static GNN has completed its forward pass. Specifically, the entire model forward pass at each step is given by:

$$\begin{cases} \tilde{H}_t^{(k)} & = GNN(H_t^{(k-1)}) \\ H_t^{(k)} & = UPDATE(H_t^{(k-1)}, \tilde{H}_t^{(k)}) \end{cases} \tag{4.3}$$

Where the 'UPDATE' method can be a moving average, an MLP that takes the concatenation of $H_t^{(k-1)}$ and $\tilde{H}_t^{(k)}$ as input, or an RNN (GRU) unit. Figure 4.3 details the forward algorithm when 'UPDATE' = MLP.

Note that this implementation only requires the current snapshot, $H_t^{(k-1)}$and$\tilde{H}_t^{(k)}$ to be in-memory at each time.

Live-Update Evaluation Setting

Instead of using a temporal split, the authors propose to evaluate ROLAND at each time step t by fine tuning the model up to time step $t-1$ and then using the ground-truth labels at t to evaluate the model. The process is then repeated for time step $t+1$. Since data distributions may be not stationary, the authors argue that this way of evaluating TGNNs is more sound.

Algorithm 1 ROLAND GNN forward computation

Input: Dynamic graph snapshot G_t, hierarchical node state H_{t-1}
Output: Prediction y_t, updated node state H_t

1: $H_t^{(0)} \leftarrow X_t$ {Initialize embedding from G_t}
2: **for** $l = 1, \ldots, L$ **do**
3: $\quad \tilde{H}_t^{(l)} = \textsc{Gnn}^{(l)}(H_t^{(l-1)})$
4: $\quad H_t^{(l)} = \textsc{Update}^{(l)}(H_{t-1}^{(l)}, \tilde{H}_t^{(l)})$
5: $y_t = \textsc{Mlp}(\textsc{Concat}(\mathbf{h}_{u,t}^{(L)}, \mathbf{h}_{v,t}^{(L)})), \forall (u, v) \in E$

Figure 4.3: The forward computation of ROLAND. *Courtesy of [63].*

Training via Meta-Learning

Since temporal data is often affected by seasonality, training the model at time step t by fine tuning the model trained at time step $t-1$ may not be optimal. Therefore, the authors take a meta-learning approach from [25] to obtain a parameter set that has the property of being a "good initialization" for fine-tuning in every time step. In the following, we describe the original form of meta-learning [14], one of its approximation [25] and how it has been adopted by the authors of ROLAND.

Informally, [14] describes meta-learning as, given a model f_θ, the task of finding a parameter set θ_0^* such that, after K steps of gradient descent, the model performance on the test set is maximized.

Concretely, we define a task $\mathcal{T}$ as a triple:

$$\mathcal{T} := (D, D^{'}, \mathcal{L})$$

Where $D = (X^j, y^j)_{j=1}^{N}$ is the training set, $D' = (X^{j'}, y^{j'})_{j'=1}^{N'}$ is the test set and $\mathcal{L}$ is the training loss. Given a model f_θ, we denote the action of K steps of gradient descent under task $\mathcal{T}$ with initial conditions θ_0 as:

$$f_{\theta'} := U^K(\mathcal{T}, f_{\theta_0})$$

Now consider a set of tasks $\mathcal{D} \sim p(\mathcal{T})$ that is split into a set of training tasks $\mathcal{D}_\mathcal{T}$ and a set of test tasks $\mathcal{D}_\mathcal{T}^{'}$:

$$\mathcal{D} = \mathcal{D}_\mathcal{T} \cup \mathcal{D}_\mathcal{T}^{'} = \{\mathcal{T}_i\}_{i=1}^{T} \cup \{\mathcal{T}_i^{'}\}_{i=T+1}^{T+\bar{T}}$$

Where:

$$\mathcal{T}_i := \{D_i, D_i^{'}, \mathcal{L}_i\}, D_i := \{(X_i^j, y_i^j)\}_{j=1}^{N_i}, D_i^{'} = \{(X_i^{j'}, y_i^{j'})\}_{j'=1}^{N_i^{'}}$$

And, as before, we define:

$$f_{\theta_i'} := U^K(\mathcal{T}_i, f_{\theta_0})$$

The meta-task consists in finding θ_0^* such that, after K steps of gradient descent, $f_{\theta'}$ minimizes the losses over the test sets i.e. the following quantity:

$$\mathcal{L}_{meta}(\theta_0) := \sum_{i=1}^{T} \mathcal{L}_i(D_i^{'}, f_{\theta_i^{'}})$$

is minimized with respect to θ_0.

Our purpose is then to obtain a formula for updating θ_0 by looking at the data. For simplicity, we perform meta-minimization using vanilla Gradient Descent, although we will assume that the results we get are also valid when SGD and/or Adam are employed. Simple gradient descent implies that:

$$\theta_0 \leftarrow \theta_0 - \nabla_{\theta_0} \mathcal{L}_{meta}(\theta_0)$$

Therefore, we need to compute $\nabla_{\theta_0} \mathcal{L}_{meta}(\theta_0)$. Assuming *full-batch* everywhere w.l.o.g.:

$$\begin{aligned} \nabla_{\theta_0} \mathcal{L}_{meta}(\theta_0) = \nabla_{\theta_0} \sum_{i=1}^{T} \mathcal{L}_i(D_i^{'}, f_{\theta_i^{'}}) = \nabla_{\theta_0} \sum_{i=1}^{T} \sum_{j=1}^{N_i^{'}} \ell_i(y_i^{j'}, U^K(\mathcal{T}_i, f_{\theta_0})(X_i^{j'})) \\ = \sum_{i=1}^{T} \sum_{j=1}^{N_i^{'}} \frac{\partial \ell_i}{\partial \hat{y}}(y_i^{j'}, f_{\theta_i^{'}}(X_i^{j'})) \underbrace{\nabla_{\theta_0}[U^K(\mathcal{T}_i, f_{\theta_0})]}_{\text{differentiate GD itself}}(X_i^{j'}) \end{aligned} \tag{4.4}$$

Thus, the quantity we need to evaluate in order to have an update rule for the meta-task is $\nabla_{\theta_0}[U^K(\mathcal{T}_i, f_{\theta_0})]$.

To this end, we notice that:

$$\begin{aligned} \theta_1 &\leftarrow \theta_0 - \nabla_\theta \mathcal{L}_\rangle(\theta_0) \\ \theta_2 \leftarrow \theta_1 - \nabla_\theta \mathcal{L}_\rangle(\theta_1) &= \theta_0 - \nabla_\theta \mathcal{L}_\rangle(\theta_0) - \nabla_\theta \mathcal{L}_\rangle(\theta_0 - \nabla_\theta \mathcal{L}_\rangle(\theta_0)) \\ &\cdot \\ &\cdot \\ &\cdot \\ \theta_K &\leftarrow R_i^K(\theta_0) \end{aligned} \tag{4.5}$$

Where:

$$\begin{cases} R_i^0(\theta_0) & := \theta_0 \\ R_i^k(\theta_0) & := R_i^{k-1}(\theta_0) - \nabla_\theta \mathcal{L}_\rangle(R_i^{k-1}(\theta_0)) \end{cases} \tag{4.6}$$

Then:

$$\nabla_{\theta_0}[U^K(\mathcal{T}_i, f_{\theta_0})](x_0) = \nabla_{\theta_0}[f(x, R_i^K(\theta_0))](x_0)$$

If $K = 1$:

$$\nabla_{\theta_0}[U^1(\mathcal{T}_i, f_{\theta_0})](x_0) = \nabla_{\theta_0}[f(x, \theta_0 - \nabla_\theta \mathcal{L}_\rangle(\theta_0))](x_0) = \nabla_\theta f(x_0, R_i^1(\theta_0))(\mathbb{I} - \mathbb{H}^\theta_{\mathcal{L}_i}(x_0, \theta_0))$$

For whatever K:

$$\nabla_{\theta_0}[U^K(\mathcal{T}_i, f_{\theta_0})](x_0) = \nabla_\theta f(x_0, R_i^K(\theta_0))(\mathbb{I} - \sum_{k=1}^{K} \mathbb{H}^\theta_{\mathcal{L}_i}(x_0, R_i^{k-1}(\theta_0))) \tag{4.7}$$

Equation (4.7) is known as *Model-Agnostic Meta-Learning* (*MAML*) framework [14].

The authors notice that dropping the second-order derivatives leads to minor performance reduction. Therefore they approximate:

$$\nabla_{\theta_0}[U^K(\mathcal{T}_i, f_{\theta_0})](x_0) \approx \nabla_\theta f(x_0, R_i^K(\theta_0)) \tag{4.8}$$

Equation (4.8) is known as the *First-Order MAML* (*FOMAML*).

A further approximation, owed to [25], allows to write the meta-update as:

$$\nabla_{\theta_0} \mathcal{L}_{meta}(\theta_0) = \nabla_{\theta_0} \sum_{i=1}^{T} \mathcal{L}_i(D_i^{'}, f_{\theta_i^{'}}) \approx \frac{\epsilon}{T} \sum_{i=1}^{T} (\theta_i^{'} - \theta_0) \tag{4.9}$$

Which for $K = 1$ reduces to *joint training* but for $K > 1$ the authors prove that the approximation is still valid.

Equation (4.9) is known as the *Reptile Algorithm* (*RA*).

Algorithm 3 ROLAND training algorithm

Input: Graph snapshot G_t, link prediction label y_t, hierarchical node state H_{t-1}, smoothing factor α, meta-model $\text{GNN}^{(meta)}$
Output: Model GNN, updated meta-model $\text{GNN}^{(meta)}$

1: $\text{GNN} \leftarrow \text{GNN}^{(meta)}$
2: Move GNN, G_t, H_{t-1} to GPU
3: **while** $\text{MRR}_t^{(val)}$ is increasing **do**
4: $\quad H_t, \hat{y}_t \leftarrow \text{GNN}(G_t, H_{t-1}), \hat{y}_t = \hat{y}_t^{(train)} \cup \hat{y}_t^{(val)}$
5: $\quad$ Update GNN via backprop based on $\hat{y}_t^{(train)}, y_t^{(train)}$
6: $\quad \text{MRR}_t^{(val)} \leftarrow \text{EVALUATE}(\hat{y}_t^{(val)}, y_t^{(val)})$
7: Remove GNN, G_t, H_{t-1} from GPU
8: $\text{GNN}^{(meta)} \leftarrow (1-\alpha)\text{GNN}^{(meta)} + \alpha\text{GNN}$

Figure 4.4: The ROLAND training algorithm. *Courtesy of [63].*

ROLAND maps the STG setting to a meta-learning scenario, by interpreting each snapshot as a different task $\mathcal{T}_i$. We therefore obtain the training algorithm in Figure 4.4, where $GNN^{(meta)}$ is the optimal initialization we are looking for (the θ_0 in $\theta_0 \leftarrow \theta_0 - \nabla_\theta \mathcal{L}_\rangle(\theta_0)$) and α replaces $\frac{\epsilon}{T}$.

The authors conclude the paper by proving the superiority of ROLAND in link prediction tasks over STGs.

4.1.3 Temporal Embedding methods (TGAT)

Temporal Embedding methods are event-based TGNNs that deal with time by encoding it. The representative architecture we are going to explore is the *Temporal Graph Attention Network* (*TGAT*) [47].

At high level, the core idea of [47] is to compute time-dependent node embeddings $\vec{h}_v^{(L)}(t)$ by aggregating over the entire time-neighborhood of v up to time t i.e. over all nodes that are connected to v at time t. During aggregation, the embedding of the i-th neighbor u_i is concatenated with the embedding of the time difference $t - t_i$, where t_i is the instant u_i connected

to v. Aggregation si performed via a self-attention scheme borrowed from the Transformer architecture [19]. In the following, we will discuss time encodings and self attention. Finally, we will put them together to describe TGAT.

Self-Attention

The authors of [47] argue in favor of a self-attention ([19]) based aggregation strategy due to its similarity to the successful GAT architecture (see Section 2.1.1), its ability to be parallelized and its interpretability. Self-attention was initially developed for sequence-based data, where the position of each token $\vec{z}_k$ in each sequence was specified via (eventually learnt) position encoding vectors $\vec{p}_k$. Position encodings are shared across sequences. Therefore, the input embeddings are:

$$Z = [\vec{z}_1||\vec{p}_1, ..., \vec{z}_l||\vec{p}_l]$$

Where $||$ denotes concatenation. Then, self-attention takes the form:

$$Attn(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{(d)}}\right)V$$

Where $Q, K \text{and} V$ are learnt *projections* of Z i.e. $Q, K, V = ZW_{Q,K,V}$ and are respectively named *Query*, *Key* and *Value* (projections). Usually, $W_Q \text{and} W_K$ are taken as the identity. The *softmax* computes the attention weights, hence the name self-attention.

Functional Time Encodings

This paragraph is heavily inspired by [35]. We start by noticing that positional encodings would prevent us from modeling continuous time. Therefore, we need a way to substitute them with continuous time-encodings.

Due to the dot-product nature of self-attention (see Section 4.1.3 above), we start by defining a *kernel* $\mathcal{K}$ such that temporal correlations are extracted via dot products:

$$\mathcal{K}(t_1, t_2) = \vec{\Phi}(t_1)\vec{\Phi}(t_2)$$

Where $\vec{\Phi}$ is the time encoding we are looking for. Moreover, since absolute timestamps carry little information, our purpose is to encode time differences $\Delta t = t_i - t_j$. Therefore, we require that $\mathcal{K}$ only depends on time differences:

$$\mathcal{K}(t_1, t_2) \overset{!}{=} \psi(t_2 - t_1)$$

The latter property is known as *translation invariance*.

To obtain a functional form for $\vec{\Phi}$, we present to approaches that lead to equally valid and similar results, namely Bochner's Theorem and Mercer's Theorem.

Theorem 4.1.1 (Bochner) *Given continuous, translation invariant and positive definite $\mathcal{K}(t_1, t_2)$, then there exists non-negative measure $\rho(\omega)$ on $\mathbb{R}$ s.t. :*

$$\mathcal{K}(t_1, t_2) = \mathcal{F}[\rho](t_1 - t_2)$$

Or equivalently:

$$\begin{aligned}\mathcal{K}(t_1, t_2) &= Re\left(\int_{\mathbb{R}} e^{i\omega(t_1 - t_2)}\rho(\omega)d\omega\right) = E_{\rho(\omega)}[cos(\omega(t_1 - t_2))] \\ &= E_{\rho(\omega)}[cos(\omega t_1)cos(\omega t_2) + sin(\omega t_1)sin(\omega t_2)]\end{aligned}$$

We may approximate the integral via Monte-Carlo extractions $\{\omega_i\}_{i=1}^{d} \sim \rho(\omega)$ by setting:

$$\Phi(t) = \sqrt{\frac{1}{d}}[cos(\omega_1 t), sin(\omega_1 t), ..., cos(\omega_d t), sin(\omega_d t)]$$

For the same reasons explained in Section 2.2.5, the sampling operation is not differentiable and therefore need to be substituted with an extraction $\vec{\epsilon} \sim p(\vec{\epsilon})$. The latter may then be combined with one of the following during training:

- Reparametrization trick $\vec{\omega} = \vec{\mu} + \vec{\epsilon} \odot \vec{\sigma}$;

- Learning of inverse CDF of ρ, $F^{-1} = g_\theta(\epsilon)$ using NN, normalizing flows, etc...

Or, if $\vec{\epsilon}$ is sampled *once* during forward pass, then the $\{\omega_i\}_{i=1}^{d}$ might be learnt directly.

As stated before, and alternative to Bochner's Theorem is Mercer's Theorem, which would lead to:

$$\Phi(t) = [\sqrt{c_1(\omega)}, ..., \underbrace{\sqrt{c_{2j}(\omega)}cos(\frac{j\pi t}{\omega})}_{\text{even element}}, \underbrace{\sqrt{c_{2j+1}(\omega)}sin(\frac{j\pi t}{\omega})}_{\text{odd element}}, ...]$$

Where:

- *sin*s and *cos*s are the Fourier basis; - $c_i(\omega)$ are the corresponding coefficients that can be optimized.

The authors of [35] proceed to prove the effectiveness of their embeddings over multiple datasets and tasks.

Temporal Graph Attention Network

Given a node v, we denote its interactions with node j lasting from t_s to t_f with $e_{uv}(t_s, t_f)$. Moreover, we denote:

$$T_s(e_{uv}(t_s, t_f)) = t_s$$

The temporal neighborhood of nodes which interacted with v before time t is:

$$N(v, t) = \{u | T_s(e_{uv}(t_s, t_f)) \leq t\}$$

We discard multiple interactions.

As anticipated, we use Bochner's/Mercer's theorem to substitute positional encodings with functional time encodings. As before (see Section 2.1.1) we denote with $\vec{\epsilon}_{uv} \in \mathcal{E}$ the features of the temporal edge $e_{uv}(t_s, t_f)$, and with $\vec{\Phi}$ the functional time encoding derived from Bochner or Mercer's theorem. For reference node v, at the l-th layer we obtain the embeddings:

$$\begin{aligned} Z^{(l-1)}(t) := [\vec{h}_v^{(l-1)}(t) || \vec{x}_{(v,v)} || \Phi(0), \vec{h}_{u_1}^{(l-1)}(t_1) || \vec{x}_{(u_1,v)} || \Phi(t - t_1), ... \\ , \vec{h}_{u_1}^{(l-1)}(t_N) || \vec{x}_{(u_N,v)} || \Phi(t - t_N)]^t \end{aligned} \tag{4.10}$$

Therefore the projected matrices are given by:

$$\begin{aligned} \vec{q}_v^{(l-1)}(t)^t := \vec{Z}_v^{(l-1)}(t)W_Q := Z^{(l-1)}(t)[0,:]W_Q \\ K^{(l-1)}(t) := Z^{(l-1)}(t)[1:N,:]W_K \\ V^{(l-1)}(t) := Z^{(l-1)}(t)[1:N,:]W_V \end{aligned} \tag{4.11}$$

Thus, the temporal neighborhood is embedded using self-attention:

$$\tilde{\vec{h}}_v^{(l)}(t) = \sum_{u \in N(v;t)} \underbrace{softmax\left(\frac{\vec{q}_v^{(l-1)}(t)^t \vec{K}_u^{(l-1)}(t)}{\sqrt{d}}; N(v;t)\right) \vec{V}_u^{(l-1)}(t)}_{\tilde{\vec{h}}_{uv}^{(l)}(t)}$$

To complete the TGAT layer, the embedding of the temporal neighborhood is concatenated to the raw features of the target node and passed to an MLP:

$$\vec{h}_v^{(l)}(t) = \text{ReLU}([\tilde{\vec{h}}_v^{(l)}(t)||\vec{h}_v^{(l-1)}(t)]W_0^{(l)} + \vec{b}_0^{(l)})W_1^{(l)} + \vec{b}_1^{(l)}$$

Now, the TGAT layer is completely specified.

In analogy to the GAT architectures, also TGAT admits a multiple-head configuration:

$$\vec{h}_v^{(l)}(t) = \text{ReLU}([\tilde{\vec{h}}_v^{(l,1)}(t)||...||\tilde{\vec{h}}_v^{(l,K)}(t)||\vec{h}_v^{(0)}(t)]W_0^{(l)} + \vec{b}_0^{(l)})W_1^{(l)} + \vec{b}_1^{(l)}$$

.

4.1.4 Temporal Neighborhood methods (TGN)

Temporal Neighborhood methods keep a *memory* vector $\vec{s}_v(t)$ for each node v that gets updated whenever an event happen to that node (its features change or it receives/outputs an edge). But, this mechanism alone would leave a node's embedding unchanged during the periods in which the node is inactive. Therefore, a time-aware aggregation step is employed (e.g. that of section 4.1.3) to compute the indirect effects on a nodes' embedding due to events that concern its neighborhood. The representative architecture we are going to discuss is *Temporal Graph Network* or *TGN* [44].

In TGN, the memory $\vec{s}_v(t)$ will be used to update nodes' features before aggregation:

$$\vec{h}_v^{(0)}(t) = \vec{x}_v(t) + \vec{s}_v(t)$$

Where the memory $\vec{s}_v(t)$ encodes all events prior to t that involve v. So let us now describe how is $\vec{s}_v(t)$ computed explicitly. We distinguish events into two categories: node-related events $\vec{x}_v(t)$ (node creation, node feature update, node deletion) and edge-related events $e_{uv}(t)$ (edge creation, edge deletion).

A feature update concerning node v will produce a *message* for node v according to:

$$\vec{m}_v(t) = \overrightarrow{msg_n}(\vec{s}_v(t^-), t, \vec{x}_v(t))$$

Where t^- is the last time $\vec{s}_v$ was updated. Node v's deletion at time t would simply remove v from every temporal neighborhood past time t.

An *edge creation* event would instead produce two messages, one for the source and the other for the target node:

$$\begin{cases} \vec{m}_u(t) & = \overrightarrow{msg_s}(\vec{s}_u(t^-), \vec{s}_v(t^-), \Delta t, e_{uv}(t)) \\ \vec{m}_v(t) & = \overrightarrow{msg_d}(\vec{s}_v(t^-), \vec{s}_u(t^-), \Delta t, e_{uv}(t)) \end{cases} \tag{4.12}$$

An *edge deletion* event would involve different instances of the same functional forms:

$$\begin{cases} \vec{m}_u(t) & = \overrightarrow{msg'_s}(\vec{s}_u(t^-), \vec{s}_v(t^-), \Delta t, e_{uv}(t)) \\ \vec{m}_v(t) & = \overrightarrow{msg'_d}(\vec{s}_v(t^-), \vec{s}_u(t^-), \Delta t, e_{uv}(t)) \end{cases} \tag{4.13}$$

The function $\overrightarrow{msg_n}, \overrightarrow{msg_s}, \overrightarrow{msg_d}, \overrightarrow{msg'_s}, \overrightarrow{msg'_d}$ are all potentially learnable.

If events are processed in batches, it could be that several messages are outputted for the same node in the same batch. TGN prescribes to aggregate such messages using a message aggregation function:

$$\vec{m}_v(t) = \overrightarrow{agg}(\vec{m}_v(t_1), ..., \vec{m}_v(t_b))$$

Once we have just one message for all active nodes, we may update its memory:

$$\vec{s}_v(t) = \overrightarrow{mem}(\vec{m}_v(t), \vec{s}_v(t^-))$$

Commonly used functional forms are:

- $\overrightarrow{msg_s}, \overrightarrow{msg_d}, \overrightarrow{msg_n}$: $\overrightarrow{CONCAT}$
- $\overrightarrow{agg}$: most recent message or mean
- $\overrightarrow{mem}$: LSTM or GRU

As we said earlier, this update mechanism would lead to embedding staleness due to event sparsity. Therefore, a time-aware aggregation mechanism is provided so that a node's embedding is also updated when one of its neighbors takes part in an event. We list below a few possibilities for the *time-aware network aggregation* layer:

- Identity: $\vec{z}_v(t) = \vec{s}_v(t)$
- Time projection: $\vec{z}_v(t) = (1 + \Delta t \vec{w}) \odot \vec{s}_v(t)$
- TGAT:

$$\begin{cases} \vec{z}_v(t) & = \vec{h}_v^L(t) \\ \vec{h}_v^{(l+1)}(t) & = \text{TGAT}(\vec{h}_v^{(l)}(t), G_T^E(<t)) \\ \vec{h}_v^{(0)}(t) & = \vec{x}_v(t) + \vec{s}_v(t) \end{cases} \tag{4.14}$$

- Temporal Graph Sum:

$$\begin{cases} \vec{h}_v^{(l)}(t) & = W_2^{(l)}(\vec{h}_v^{(l-1)}(t) || \tilde{\vec{h}}_v^{(l)}(t)) \\ \tilde{\vec{h}}_v^{(l)}(t) & = ReLU\left(\sum_{u \in \mathcal{N}_v([0,t])} W_1^{(l)}\left(\vec{h}_u^{(l-1)}(t) || e_{uv} || \Phi(t - t_u)\right)\right) \end{cases} \tag{4.15}$$

Figure 4.5 is an illustration of TGN's information flow.

4.2 Tasks over Temporal graphs

Example tasks over temporal graphs are the temporal analogue of those already mentioned for static graphs: Temporal Node Classification, Temporal

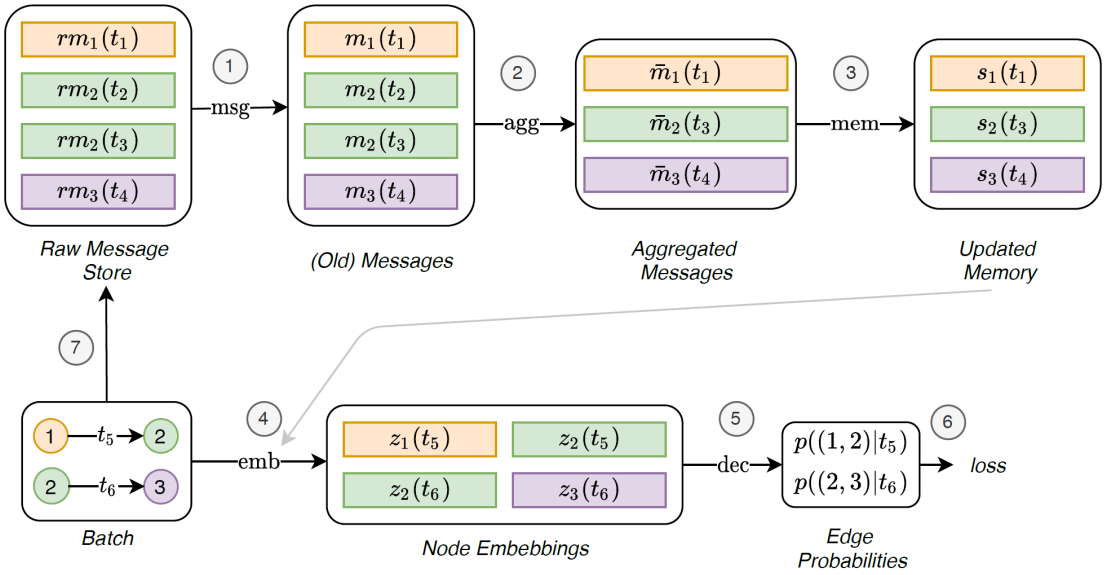


Figure 4.5: Schematic representation of TGN's information flow. The "Raw Message Stores" pre-caches the input needed to message functions for each new batch of events. Then messages are computed (1) and aggregated (2), and finally memories are updated (3). Memories are used to update raw nodes' features, which are later fed to the time-aware network aggregation layer (4). The obtained embeddings are used to perform downstream tasks such as link prediction (5 and 6). *Courtesy of [44].*

Edge Classification, Temporal Graph Classification, Temporal Link Prediction, Event Time Prediction, Temporal Node Clustering, Temporal Graph Clustering. Since they will be later useful, we report the definitions of Temporal Node Classification and Temporal Link Prediction as in [70]:

Definition 4.2.1 (Temporal Node Classification) *Given a Temporal Graph $G_T = (V, E, V_T, E_T)$, the temporal node classification task consists in learning the function*

$$f_{NC} : V \times \mathbb{R}^+ \to C$$

Which maps each node to a class $C \in C$, at a time $t \in \mathbb{R}^+$.

Definition 4.2.2 (Temporal Link Prediction) *Given a Temporal Graph $G_T = (V, E, V_T, E_T)$, the temporal link prediction task consists in learning the function*

$$f_{LP} : V \times V \times \mathbb{R}^+ \to [0,1]$$

Which predicts the probability that, at a certain time, there exists an edge between two given nodes.

Tasks can be carried out in different settings:

Definition 4.2.3 *Assume that a model is trained on a set of $n \geq 1$ temporal graphs $\mathcal{G}_T := \{G_T^i := (V_i, E_i, X_i^V, X_i^E), i = 1, ..., n\}$. Moreover, let:*

$$T_e^{all} := \max_{i=1,...,n} T_e(G_T^i), \qquad V^{all} := \cup_{i=1}^n V_i, E^{all} := \cup_{i=1}^n E_i$$

Be the final timestamp and the set of all nodes and edges in the training set. Then, we have the following settings:

- *Transductive learning: inference can only be performed on $v \in V^{all}, e \in E^{all}$ or $G_T \subseteq_V G_T^i$ with $G_T^I \in \mathcal{G}_T$*
- *Inductive learning: inference can only be performed also on $v \notin V^{all}, e \notin E^{all}$ or $G_T \not\subseteq_V \forall i = 1, ..., n$*
- *Past prediction: inference is performed for $t \leq T_e^{all}$*
- *Future prediction: inference is performed $t > T_e^{all}$*

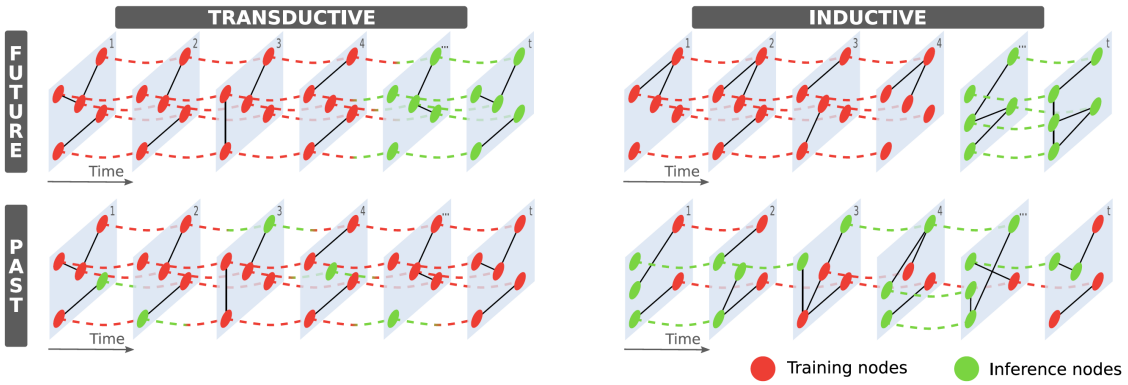


Figure 4.6: The four possible settings. Dashed lines connect instances of the same temporal nodes at different time steps. *Courtesy of [70].*

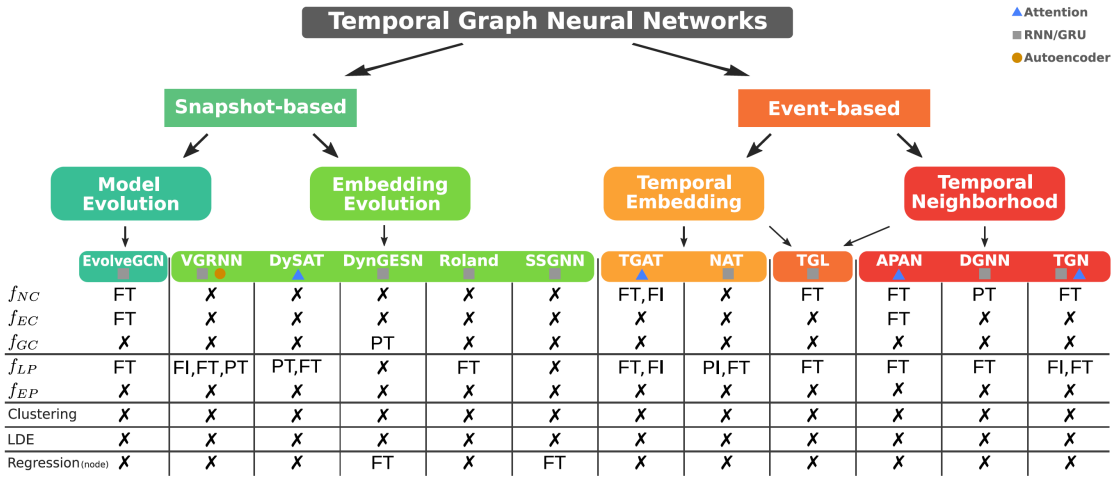


Figure 4.7: A taxonomy of TGNN architectures, and the (tasks,setting) pairs they're able to tackle. *Courtesy of [70].*

Figure 4.6 summarizes tasks' classification.

[70] complements the review with the table in Figure 4.7, which organizes TGNN architecture with respect to the taxonomy we described above, and also reports on what setting may each model tackle each task (Future Transductive FT, Future Inductive FI, Past Transductive PT, Past Inductive PI).

Despite their successes, it is our opinion that TGNNs have a few shortcomings:

- Conceptually not very sound: except TGAT, the other architectures are a mix of recursive, convolutional and meta-learning layers. They look more like adapted, ad-hoc rather than natural architectures for the data type we are considering;
- Due to their heterogeneity, it is difficult to come up with a mathematical and software framework to encompass all of them. This causes difficulties in implementation and training;
- Outside of the datasets they've been expertly tuned on, these architectures sometimes strive to get good performance on other datasets. See e.g. their underwhelming performance on the "Dynamic Node Property Prediction" task on TGB (TGB is a collection of model benchmarks of tasks over temporal graphs), and also they are outperformed by simple baselines in temporal node classification tasks [74] .

In essence, TGNNs constitute an attempt to design architectures that may ingest temporal graphs. We think it would be worth exploring the inverse avenue: instead of "lifting" GNNs to the temporal domain, we set out to "reducing" dynamic graphs to static networks (with specific techniques to preserve temporal information) and then apply standard GNNs over them. Of course, this would also imply mapping the task on the temporal graph to the static graph. Such an approach may have the advantage of being simpler to implement.

In the following, we will explore temporal information-preserving static representations, and finally present an early result on a temporal node regression task.

4.3 Time-Preserving Static Representations

Just as applying static GNNs to temporal graphs leads to inferior performances (see e.g. [47]), we also expect any naive reductions of dynamic graphs to static networks to imply information loss too. We will name with *Static Representation* any technique to map a temporal graph to a static one. We will further denote *time-preserving* any static representation that admits a unique, bijective inverse transformation from the static network to the temporal one that originated it. For instance, a simple time-preserving static representation would be to turn the temporal graph into a multigraph where edge timestamps are concatenated to edge feature vectors. Indeed, there is a

albeit small body of literature that attempts to design time-preserving static representations that we summarize here.

An article on this topic is [42] . Given a temporal undirected graph $G = (V, E)$, where $E \subseteq \{(\{u, v\}, t) | u, v \in V \wedge u \neq v\}$, they develop three techniques to reduce a temporal (undirected) graph to a static graph:

Definition 4.3.1 (Reduced Graph Representation) *The Reduced Graph Representation (RR) consists in converting the temporal graph to a static graph where, for each temporal edge, only its earliest availability is conserved as an edge label.*

This approach loses much of the temporal information.

Definition 4.3.2 (Directed Line Graph Expansion) *The Directed Line Graph Expansion (DLGE) consists in substituting each temporal edge with two vertices, each labelled with one of the two possible edge's directions and its availability time. More precisely, the Directed Line Graph Expansion of G is a directed static graph $DL(G) = (V', E')$ where V' and E' are respectively defined by:*

- $(\{u, v\}, t) \in E \implies n^t_{\overrightarrow{uv}}, n^t_{\overrightarrow{vu}} \in V'$;
- $(n^t_{\overrightarrow{uv}}, n^s_{\overrightarrow{xy}}) \in E' \iff v = x \wedge s > t$

Consequently, only vertices that corresponded to adjacent edges are linked with an edge directed according to the nodes' timestamps (from the smaller to the higher). This representation does not lose any information, and may be extended to directed graphs by only creating one vertex instead of two for every edge in the temporal graph, namely the one corresponding to the direction of the vertex.

A stochastic version of this algorithm is also presented, to make up for the $O(|E|^2)$ complexity of $DL(G)$.

Definition 4.3.3 (Static Expansion) *The Static Expansion (SE) consists in creating a static representation $SE(G) = (U, E_{SE(G)})$ where for each temporal edge $(\{u, v\}, t) \in E$ four time-vertices are added to U, namely $\{(u, t), (v, t), (u, t + 1), (v, t + 1)\}$. Moreover, $E_{SE(G)} = E_N \cup E_{W_1} \cup E_{W_2}$ where:*

- $E_N := \{((u, t), (v, t + 1)), ((u, t + 1), (v, t)) | (\{u, v\}, t) \in E\}$ *contributes to $E_{SE(G)}$ directed edges between time vertices that are connected in the temporal graph. Since edges in G are undirected, we need to add one*

edge to $E_{SE(G)}$ for each of the two directions. These edges will be named structural;

- $E_{W_1} := \{((w, i+1), (w, j))|(w, i+1), (w, j) \in U \wedge i, j \in T(w) \wedge \tau_w(i)+1 = \tau_w(j) \wedge i + i < j\}$ *and* $E_{W_2} := \{((w, i), (w, j))|(w, i), (w, j) \in U \wedge i, j \in T(w) \wedge \tau_w(i) + 1 = \tau_w(j) \wedge i+i < j\}$ *contribute to $E_{SE(G)}$ edges between the two time vertices associated to each node w that get created for each temporal edge incident on it at time i, namely (w, i) and $(w, i+1)$, and the time vertex associated to node w which is incident to the next edge in order of availability (i.e. at time j). $\tau_w(i)$ is the time-ordered index of availability time i in the set of availability times of the temporal edges incident to w, namely $T(w)$. These edges will be named temporal.*

Finally, the authors apply Weisfeiler–Leman and an RW-based algorithm to the static representation, and prove that *DLGE* and *SE* outperform *RR* and other baselines in complex temporal graph classification tasks using an SVM. Crucially, this results show that the above static representations are compatible with the WL-kernel, which is just a non-trainable version of GNNs. Although this is proven only for dynamic graph classification task, it motivated us to try the same static representations to also map other tasks. The same experiments have ben run also using GNNs [41], although they didn't outperform kernels.

The authors of [31] summarize and extend the work of [23] and [18]. They extend the definition of Directed Line Graph Expansion to directed networks by modifying its conditions to:

- $((u, v), t) \in E \implies n^t_{\overrightarrow{uv}} \in V'$;
- $(n^t_{\overrightarrow{uv}}, n^s_{\overrightarrow{xy}}) \in E' \iff v = x \wedge s > t$

They also define a thresholded version where, given a time interval Δt, the conditions become:

- $((u, v), t) \in E \implies n^t_{\overrightarrow{uv}} \in V'$;
- $(n^t_{\overrightarrow{uv}}, n^s_{\overrightarrow{xy}}) \in E' \iff v = x \wedge s > t \wedge s - t < \Delta t$

We would also propose an extension of Static Expansions to directed networks (where we also drop the requirement that edge traversal costs 1 unit of time):

Definition 4.3.4 (Directed Static Expansion) *The Directed Static Expansion (SE) consists in creating a static representation $DSE(G) = (U, E_{SE(G)})$*

where for each directed temporal edge $((u, v), t) \in E$ *two time-vertices are added to* U*, namely* $\{(u, t), (v, t)\}$*. Moreover,* $E_{SE(G)} = E_N \cup E_W$ *where:*

- $E_N := \{((u, t), (v, t))|(\{u, v\}, t) \in E\}$ *contributes to* $E_{SE(G)}$ *the directed edges between time vertices that are connected in the temporal graph. These edges will be named structural;;*
- $E_W := \{((w, i), (w, j))|(w, i), (w, j) \in U \wedge i, j \in T(w) \wedge \tau_w(i) + 1 = \tau_w(j) \wedge i + i < j\}$ *contribute to* $E_{SE(G)}$ *edges between the time-vertex associated to each temporal node* w *that get created for each temporal edge incident on it at time* i*, namely* (w, i)*, and the time vertex associated to node* w *which is incident to the next edge in order of availability (i.e. at time* j*).* $\tau_w(i)$ *is the time-ordered index of availability time* i *in the set of availability times of the temporal edges incident to* w*, namely* $T(w)$*. These edges will be named temporal.*

Therefore, all things considered, we may group time-preserving static representations in two categories: *Directed Line Graph Expansion-like* (*DLGE-like*) and *Static Expansions-like* (*SE-like*).

As we said earlier, once the dynamic network is mapped to its static representations, also the task over it must be ported. While mapping graph classification is trivial (as [42] and [41] do), mapping Dynamic Node Classification and Dynamic Link Prediction tasks is less so. We found that the easier class of Static Representations to work with are the *Static Expansions*. Since all nodes and edges are preserved, mapping tasks is trivial. In the next section, we showcase an experiment where we managed to beat TGNNs in a dynamic node classification task employing standard GNNs over a Static Expansion of an STG.

4.4 Dynamic Node Classification using Static Representations

As a demonstration, we chose to tackle the task associated to the TGBN-Trade dataset on TGB.

The dataset is a directed temporal network constituted by 255 nodes and 468'245 temporal edges. Nodes represent nations, while edges represent agricultural trades. Snapshots are yearly, and edge weights represent the total amount of trade between the two nations it connects during the full year. In

total, there are 30 snapshots.

The task is to predict, each year staring from 2017, the fraction of trade each nation had with all the others, that is, it amounts to predicting, for each node in each snapshot, 255 real numbers between 0 and 1. Crucially, we notice that this setting is equivalent to a 255-class node classification, where the model outputs a probability for each of the classes, if one uses Pytorch's CrossEntropyLoss with probabilities as targets, as explained in the referenced documentation (even though this is an approximation).

We implemented the DSE static representation (Definition 4.3.4). Notice that we may interpret the resulting static network as heterogeneous, since edges belonging to E_N have ha different meaning than those belonging to E_W. This will be reflected by the GNN architecture, for which we used a 2-layer heterogeneous GNN defined by:

$$\vec{h}_v^{(k+1)} = \text{ReLU}\left[\left(\sum_{u \in N_{E_N}(v)} W_{E_N}^{neigh}\vec{h}_u^{(k)} + W_{E_N}^{self}\vec{h}_v^{(k)}\right) + \left(\sum_{u \in N_{E_W}(v)} W_{E_W}^{neigh}\vec{h}_u^{(k)} + W_{E_W}^{self}\vec{h}_v^{(k)}\right)\right]$$

The node classification task is bijectively translated from the temporal to the static domain by constructing a dictionary that maps pairs (node,time) to their time-node aliases in the static representation.

The dataset was split into train/val/test according to a temporal split following the rules of the benchmark. And, since there were no node features and the temporal nature of the split made it impossible to use 'torch.nn.Embedding's, we opted for a one-hot encoding of the time-nodes. The metric used is the NDCG10, and we employed the Adam optimizer, full batch, with $lr = 0.005$ for 1000 epochs with early stopping.

Our approach achieved higher performance with respect to all TGNNs, although is still lower than a moving average (MA) that, due to the closely arithmetic relation between the features and the labels, outperforms all other algorithms (Table 4.1).

Model	NDCG@10
Ours	0.549
DygFormer	0.388
TGN	0.374
DyRep	0.374
MA	0.823

Table 4.1: Results on TGBN-Trade.

Chapter 5

Conclusions

In this thesis, we advanced the technologies prospectively needed to develop a representation learning framework for transaction networks, the long-term purposes being their deep synthetic generation and temporal subgraph matching to directly address Anti-Money Laundering. Due to their dynamic aspect, the focus was shifted to inductive models i.e. graph neural networks. After providing a comprehensive review of the topic and reproducing official node classification and link prediction tasks, we identified the two main characteristics of transaction networks: directionality and temporality. Since both these aspects need to be properly learned, we decided to explore them separately. The most straightforward way to test if a model is capable of meaningfully embedding directionality is to check if it can distinguish edges' directions and isolate bidirectional edges. To the best of our knowledge, there is no existing model able to learn both of them while retaining directed link prediction capabilities in natural settings where the positive and negative edges in the test and training sets are largely uncorrelated. Therefore, we identified the simultaneous balancing of positive/negative unidirectional/bidirectional edges as being the culprit of the issue and devised a simple training strategy that would allow a model to embed directionality while retaining most of the performance in the general directed link prediction task. Regarding temporality, we reviewed a few Temporal Graph Neural Networks, picking one model per architecture type. We decided to explore the opposite approach, which we named "Static Representations", where instead of designing architectures compatible with temporal graphs, we first compute a static representation of the dynamic graph such that temporal information is topologically preserved, and only then we apply a static GNN.

We believe these advancements take current technologies one step closer to performing proper transaction network generation and temporal subgraph matching.

5.1 Limitations

Nonetheless, our work presents the following limitations. Transaction networks are dynamic, and therefore, whatever the task, the train/validation/test split should be performed according to time. This would introduce a form of statistical imbalance between training, validation and test sets, namely node degrees, that has not been addressed in this work, not even in Section 2.5.4. Node degree imbalances would indeed be an issue, since nodes with low degree in the training set are given little importance without knowing their relevance (measured as number of events involving those nodes) in the test set. A possible fix concerning directed link prediction tasks could be to reweight each edge's contribution to the cross-entropy loss according to the degrees of the adjacent nodes. Referring to Equation (3.2), we may modify it as:

$$\mathcal{L}_T = -\sum_{e_{uv} \in E_{m+s}} w_{uv} w_{y_{uv}} \ln(\hat{p}(e_{uv})) \tag{5.1}$$

Where, having defined d_u^+ and d_u^- respectively as the out- and in-degrees of u, it may be required:

$$w_{uv} \sim \frac{1}{d_u^+ \cdot d_v^-}$$

One may also envision to train the w_{uv} and $w_{y_{uv}}$ weights by performing some form of gradient descent over the validation set, akin to a meta-learning task.

Regarding the Static Representation framework, it is clearly held back by the increased computational complexity due to the addition of structural edges. Nonetheless, although the number of such edges depends on the characteristics of the temporal network, we noticed only a negligible number in the only experiment we did. Also, we haven't tried other types of tasks such as Link Prediction.

5.2 Future Directions

Concerning generation, the immediate next step would be to train a directed Variational Graph Autoencoder, and then attempt to generate a directed network by asking the model to predict all the links among the nodes of the graph it has been trained on. Also, graph diffusion models might be extended to the directed case [72]: as anticipated, these models would also have the advantage to simultaneously generate node features, instead of requiring them as inputs as Variational Graph Autoencoders do. In the long term, temporal graph generation could be achieved by combining Variational Graph Autoencoders/Graph Diffusion Models with the Static Representation Framework. The latter must therefore be tested also under different tasks, with the additional goal of mapping subgraph matching to and from the static domain to tackle Anti-Money Laundering.

Bibliography

[1] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks", *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[2] T. Fawcett, "An introduction to roc analysis", *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, ROC Analysis in Pattern Recognition, ISSN: 0167-8655. DOI: https://doi.org/10.1016/j.patrec.2005.10.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016786550500303X.

[3] X. Li and H. Chen, "Recommendation as link prediction: A graph kernel-based machine learning approach", in *Proceedings of the 9th ACM/IEEE-CS Joint Conference on Digital Libraries*, ser. JCDL '09, Austin, TX, USA: Association for Computing Machinery, 2009, pp. 213–216, ISBN: 9781605583228. DOI: 10.1145/1555400.1555433. [Online]. Available: https://doi.org/10.1145/1555400.1555433.

[4] D. K. Hammond, P. Vandergheynst, and R. Gribonval, "Wavelets on graphs via spectral graph theory", *Applied and Computational Harmonic Analysis*, vol. 30, no. 2, pp. 129–150, 2011, ISSN: 1063-5203. DOI: https://doi.org/10.1016/j.acha.2010.04.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1063520310000552.

[5] L. Lü and T. Zhou, "Link prediction in complex networks: A survey", *Physica A: Statistical Mechanics and its Applications*, vol. 390, no. 6, pp. 1150–1170, 2011, ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2010.11.027. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S037843711000991X.

[6] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, "Weisfeiler-lehman graph kernels", *Journal of Machine Learning Research*, vol. 12, no. 77, pp. 2539–2561, 2011. [Online]. Available: http://jmlr.org/papers/v12/shervashidze11a.html.

[7] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data", in *Advances in Neural Information Processing Systems*, C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Weinberger, Eds., vol. 26, Curran Associates, Inc., 2013. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2013/file/1cecc7a77928ca8133fa24680a88d2f9-Paper.pdf.

[8] D. P. Kingma and M. Welling, *Auto-encoding variational bayes*, 2013. DOI: 10.48550/ARXIV.1312.6114. [Online]. Available: https://arxiv.org/abs/1312.6114.

[9] J. Pastor-Pellicer, F. Zamora-Martínez, S. España-Boquera, and M. J. Castro-Bleda, "F-measure as the error function to train neural networks", in *Advances in Computational Intelligence*, I. Rojas, G. Joya, and J. Gabestany, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 376–384, ISBN: 978-3-642-38679-4.

[10] M. Kivelä, A. Arenas, M. Barthelemy, J. P. Gleeson, Y. Moreno, and M. A. Porter, "Multilayer networks", *Journal of Complex Networks*, vol. 2, no. 3, pp. 203–271, Jul. 2014, ISSN: 2051-1310. DOI: 10.1093/comnet/cnu016. eprint: https://academic.oup.com/comnet/article-pdf/2/3/203/9130906/cnu016.pdf. [Online]. Available: https://doi.org/10.1093/comnet/cnu016.

[11] C. Doersch, *Tutorial on variational autoencoders*, 2016. DOI: 10.48550/ARXIV.1606.05908. [Online]. Available: https://arxiv.org/abs/1606.05908.

[12] T. N. Kipf and M. Welling, *Variational graph auto-encoders*, 2016. DOI: 10.48550/ARXIV.1611.07308. [Online]. Available: https://arxiv.org/abs/1611.07308.

[13] M. Defferrard, X. Bresson, and P. Vandergheynst, *Convolutional neural networks on graphs with fast localized spectral filtering*, 2017. arXiv: 1606.09375 [cs.LG].

[14] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks", in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17, Sydney, NSW, Australia: JMLR.org, 2017, pp. 1126–1135.

[15] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry", in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17, Sydney, NSW, Australia: JMLR.org, 2017, pp. 1263–1272.

[16] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs", in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 1025–1035, ISBN: 9781510860964.

[17] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks", in *International Conference on Learning Representations*, 2017. [Online]. Available: https://openreview.net/forum?id=SJU4ayYgl.

[18] A. Mellor, "The temporal event graph", *Journal of Complex Networks*, vol. 6, no. 4, pp. 639–659, Oct. 2017, ISSN: 2051-1329. DOI: 10.1093/comnet/cnx048. eprint: https://academic.oup.com/comnet/article-pdf/6/4/639/25451949/cnx048.pdf. [Online]. Available: https://doi.org/10.1093/comnet/cnx048.

[19] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need", in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, *et al.*, Eds., vol. 30, Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[20] M. Zaheer, S. Kottur, S. Ravanbhakhsh, B. Póczos, R. Salakhutdinov, and A. J. Smola, "Deep sets", in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17, Long Beach, California, USA: Curran Associates Inc., 2017, pp. 3394–3404, ISBN: 9781510860964.

[21] E. Bütün, M. Kaya, and R. Alhajj, "Extension of neighbor-based link prediction methods for directed, weighted and temporal social networks", *Information Sciences*, vol. 463–464, pp. 152–165, Oct. 2018, ISSN: 0020-0255. DOI: 10.1016/j.ins.2018.06.051. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2018.06.051.

[22] J. Du, S. Zhang, G. Wu, J. M. F. Moura, and S. Kar, *Topology adaptive graph convolutional networks*, 2018. [Online]. Available: https://openreview.net/forum?id=H113pWZRb.

[23] M. Kivelä, J. Cambe, J. Saramäki, and M. Karsai, "Mapping temporal-network percolation to weighted, static event graphs", *Scientific Reports*, vol. 8, no. 1, Aug. 2018, ISSN: 2045-2322. DOI: 10.1038/s41598-018-29577-2. [Online]. Available: http://dx.doi.org/10.1038/s41598-018-29577-2.

[24] Q. Liu, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Constrained graph variational autoencoders for molecule design", in *Proceedings of*

the 32nd International Conference on Neural Information Processing Systems, ser. NIPS'18, Montréal, Canada: Curran Associates Inc., 2018, pp. 7806–7815.

[25] A. Nichol, J. Achiam, and J. Schulman, *On first-order meta-learning algorithms*, 2018. arXiv: 1803.02999 [cs.LG].

[26] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks", in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ.

[27] M. Weber, J. Chen, T. Suzumura, *et al.*, *Scalable graph learning for anti-money laundering: A first look*, 2018. arXiv: 1812.00076 [cs.SI].

[28] D. P. Kingma and M. Welling, "An introduction to variational autoencoders", *Foundations and Trends® in Machine Learning*, vol. 12, no. 4, pp. 307–392, 2019. DOI: 10.1561/2200000056. [Online]. Available: https://doi.org/10.1561/2200000056.

[29] A. Pareja, G. Domeniconi, J. Chen, *et al.*, *Evolvegcn: Evolving graph convolutional networks for dynamic graphs*, 2019. arXiv: 1902.10191 [cs.LG].

[30] G. Salha, S. Limnios, R. Hennequin, V. A. Tran, and M. Vazirgiannis, *Gravity-inspired graph autoencoders for directed link prediction*, 2019. DOI: 10.48550/ARXIV.1905.09570. [Online]. Available: https://arxiv.org/abs/1905.09570.

[31] *Temporal Network Theory*. Springer International Publishing, 2019, ISBN: 9783030234959. DOI: 10.1007/978-3-030-23495-9. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-23495-9.

[32] M. Weber, G. Domeniconi, J. Chen, *et al.*, *Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics*, 2019. arXiv: 1908.02591 [cs.SI].

[33] F. Wu, A. Souza, T. Zhang, C. Fifty, T. Yu, and K. Weinberger, "Simplifying graph convolutional networks", in *Proceedings of the 36th International Conference on Machine Learning*, K. Chaudhuri and R. Salakhutdinov, Eds., ser. Proceedings of Machine Learning Research, vol. 97, PMLR, Jun. 2019, pp. 6861–6871. [Online]. Available: https://proceedings.mlr.press/v97/wu19e.html.

[34] F. Wu, T. Zhang, A. H. d. Souza, C. Fifty, T. Yu, and K. Q. Weinberger, *Simplifying graph convolutional networks*, 2019. DOI: 10.48550/ARXIV.1902.07153. [Online]. Available: https://arxiv.org/abs/1902.07153.

[35] D. Xu, C. Ruan, S. Kumar, E. Korpeoglu, and K. Achan, "Self-attention with functional time representation learning", in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[36] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?", in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km.

[37] M. Fey, J. E. Lenssen, C. Morris, J. Masci, and N. M. Kriege, "Deep graph matching consensus", in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=HyeJf1HKvS.

[38] W. Hu, M. Fey, M. Zitnik, *et al.*, "Open graph benchmark: Datasets for machine learning on graphs", in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20, Vancouver, BC, Canada: Curran Associates Inc., 2020, ISBN: 9781713829546.

[39] A. Kumar, S. S. Singh, K. Singh, and B. Biswas, "Link prediction techniques, applications, and performance: A survey", *Physica A: Statistical Mechanics and its Applications*, vol. 553, p. 124 289, 2020, ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2020.124289. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378437120300856.

[40] A. S. d. Mata, "Complex networks: A mini-review", *Brazilian Journal of Physics*, vol. 50, no. 5, pp. 658–672, Jul. 2020, ISSN: 1678-4448. DOI: 10.1007/s13538-020-00772-9. [Online]. Available: http://dx.doi.org/10.1007/s13538-020-00772-9.

[41] L. Oettershagen, N. M. Kriege, C. Morris, and P. Mutzel, "Classifying dissemination processes in temporal graphs", *Big Data*, vol. 8, no. 5, pp. 363–378, 2020, PMID: 33090027. DOI: 10.1089/big.2020.0086. eprint: https://doi.org/10.1089/big.2020.0086. [Online]. Available: https://doi.org/10.1089/big.2020.0086.

[42] L. Oettershagen, N. M. Kriege, C. Morris, and P. Mutzel, "Temporal graph kernels for classifying dissemination processes", in *Proceedings of the 2020 SIAM International Conference on Data Mining*, Society for Industrial and Applied Mathematics, Jan. 2020, pp. 496–504. DOI: 10.1137/1.9781611976236.56. [Online]. Available: https://doi.org/10.1137/1.9781611976236.56.

[43] Rex, Ying, Z. Lou, *et al.*, *Neural subgraph matching*, 2020. arXiv: 2007.03092 [cs.LG].

[44] E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein, *Temporal graph networks for deep learning on dynamic graphs*, 2020. arXiv: 2006.10637 [cs.LG].

[45] Z. Tong, Y. Liang, C. Sun, X. Li, D. Rosenblum, and A. Lim, "Digraph inception convolutional networks", in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 17 907–17 918. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/cffb6e2288a630c2a787a64ccc67097c-Paper.pdf.

[46] Z. Tong, Y. Liang, C. Sun, D. S. Rosenblum, and A. Lim, *Directed graph convolutional network*, 2020. arXiv: 2004.13970 [cs.LG].

[47] D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan, *Inductive representation learning on temporal graphs*, 2020. arXiv: 2002.07962 [cs.LG].

[48] S. J. Ahn and M. Kim, "Variational graph normalized autoencoders", in *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, ser. CIKM '21, Virtual Event, Queensland, Australia: Association for Computing Machinery, 2021, pp. 2827–2831, ISBN: 9781450384469. DOI: 10.1145/3459637.3482215. [Online]. Available: https://doi.org/10.1145/3459637.3482215.

[49] A. Hogan, E. Blomqvist, M. Cochez, *et al.*, "Knowledge graphs", *ACM Computing Surveys (Csur)*, vol. 54, no. 4, pp. 1–37, 2021.

[50] M. Starnini, C. E. Tsourakakis, M. Zamanipour, *et al.*, "Smurf-based anti-money laundering in time-evolving transaction networks", in *Lecture Notes in Computer Science*. Springer International Publishing, 2021, pp. 171–186, ISBN: 9783030865146. DOI: 10.1007/978-3-030-86514-6_11. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-86514-6_11.

[51] Y. Wang, Y.-Y. Chang, Y. Liu, J. Leskovec, and P. Li, "Inductive representation learning in temporal networks via causal anonymous walks", in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=KYPz4YsCPj.

[52] X. Zhang, Y. He, N. Brugnone, M. Perlmutter, and M. Hirn, "Magnet: A neural network for directed graphs", in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=TRDAFiwDq8A.

[53] F. Arrigo, D. J. Higham, V. Noferini, and R. Wood, "Dynamic katz and related network measures", *Linear Algebra and its Applications*, vol. 655, pp. 159–185, Dec. 2022. DOI: 10.1016/j.laa.2022.08.022. [Online]. Available: https://doi.org/10.1016/j.laa.2022.08.022.

[54] S. Bloemheuvel, J. van den Hoogen, D. Jozinović, A. Michelini, and M. Atzmueller, "Graph neural networks for multivariate time series regression with application to seismic data", *International Journal of Data Science and Analytics*, vol. 16, no. 3, pp. 317–332, Aug. 2022, ISSN: 2364-4168. DOI: 10.1007/s41060-022-00349-6. [Online]. Available: http://dx.doi.org/10.1007/s41060-022-00349-6.

[55] S. Gupta and S. Bedathur, *A survey on temporal graph representation learning and generative modeling*, 2022. arXiv: 2208.12126 [cs.LG].

[56] S. Gupta, S. Manchanda, S. Ranu, and S. Bedathur, "Tigger: Scalable generative modelling for temporal interaction graphs (to appear)", in *Proc. of the 36th AAAI Conference on Artificial Intelligence (AAAI)*, 2022.

[57] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey", *Expert Systems with Applications*, vol. 207, p. 117 921, Nov. 2022, ISSN: 0957-4174. DOI: 10.1016/j.eswa.2022.117921. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2022.117921.

[58] G. Kollias, V. Kalantzis, T. Id'e, A. C. Lozano, and N. Abe, "Directed graph auto-encoders", in *AAAI Conference on Artificial Intelligence*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247154899.

[59] D. Lin, J. Wu, Q. Xuan, and C. K. Tse, "Ethereum transaction tracking: Inferring evolution of transaction networks via link prediction", *Physica A: Statistical Mechanics and its Applications*, vol. 600, p. 127 504, 2022, ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2022.127504. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378437122003600.

[60] A. Saxena, Y. Pei, J. Veldsink, W. van Ipenburg, G. Fletcher, and M. Pechenizkiy, "The banking transactions dataset and its comparative analysis with scale-free networks", in *Proceedings of the 2021 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, ser. ASONAM '21, Virtual Event, Netherlands: Association for Computing Machinery, 2022, pp. 283–296, ISBN: 9781450391283. DOI: 10.1145/3487351.3488339. [Online]. Available: https://doi.org/10.1145/3487351.3488339.

[61] J. Suárez-Varela, P. Almasan, M. Ferriol-Galmés, *et al.*, *Graph neural networks for communication networks: Context, use cases and opportunities*, 2022. arXiv: 2112.14792 [cs.NI].

[62] H. Wu, C. Song, Y. Ge, and T. Ge, "Link prediction on complex networks: An experimental survey", *Data Science and Engineering*, vol. 7, no. 3, pp. 253–278, Jun. 2022, ISSN: 2364-1541. DOI: 10.1007/s41019-022-00188-2. [Online]. Available: http://dx.doi.org/10.1007/s41019-022-00188-2.

[63] J. You, T. Du, and J. Leskovec, *Roland: Graph learning framework for dynamic graphs*, 2022. arXiv: 2208.07239 [cs.LG].

[64] E. Altman, J. Blanuša, L. von Niederhäusern, B. Egressy, A. Anghel, and K. Atasu, *Realistic synthetic financial transactions for anti-money laundering models*, 2023. arXiv: 2306.16424 [cs.AI].

[65] D. Arrar, N. Kamel, and A. Lakhfif, "A comprehensive survey of link prediction methods", *The Journal of Supercomputing*, Sep. 2023, ISSN: 1573-0484. DOI: 10.1007/s11227-023-05591-8. [Online]. Available: http://dx.doi.org/10.1007/s11227-023-05591-8.

[66] S. Bishnoi, R. Bhattoo, Jayadeva, S. Ranu, and N. M. A. Krishnan, *Discovering symbolic laws directly from trajectories with hamiltonian graph neural networks*, 2023. arXiv: 2307.05299 [cs.LG].

[67] A. Gravina and D. Bacciu, *Deep learning for dynamic graphs: Models and benchmarks*, 2023. arXiv: 2307.06104 [cs.LG].

[68] M. Jin, H. Y. Koh, Q. Wen, *et al.*, *A survey on graph neural networks for time series: Forecasting, classification, imputation, and anomaly detection*, 2023. arXiv: 2307.03759 [cs.LG].

[69] T. J. Lakshmi and S. D. Bhavani, "Link prediction approach to recommender systems", *Computing*, Oct. 2023, ISSN: 1436-5057. DOI: 10.1007/s00607-023-01227-0. [Online]. Available: http://dx.doi.org/10.1007/s00607-023-01227-0.

[70] A. Longa, V. Lachi, G. Santin, *et al.*, "Graph neural networks for temporal graphs: State of the art, open challenges, and opportunities", *Transactions on Machine Learning Research*, 2023, ISSN: 2835-8856. [Online]. Available: https://openreview.net/forum?id=pHCdMat0gI.

[71] M. Qin and D.-Y. Yeung, "Temporal link prediction: A unified framework, taxonomy, and review", *ACM Comput. Surv.*, vol. 56, no. 4, Nov. 2023, ISSN: 0360-0300. DOI: 10.1145/3625820. [Online]. Available: https://doi.org/10.1145/3625820.

[72] C. Vignac, I. Krawczuk, A. Siraudin, B. Wang, V. Cevher, and P. Frossard, "Digress: Discrete denoising diffusion for graph generation",

in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=UaAD-Nu86WX.

[73] F. Chung, *Spectral Graph Theory* (CBMS Regional Conference Series no. 92). Conference Board of the Mathematical Sciences, ISBN: 9780821889367. [Online]. Available: https://books.google.it/books?id=YUc38_MCuhAC.

[74] F. Poursafaei, Z. Zilic, and R. Rabbany, "A strong node classification baseline for temporal graphs", pp. 648–656, DOI: 10.1137/1.9781611977172.73. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611977172.73. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/1.9781611977172.73.

Appendix A

Graph Machine Learning

In the following, we will be reviewing machine learning methods to graph tasks. These methods often involve two steps:

1. Design human-engineered features
2. Feed them to a parametric model such as XGBoost, SVMs, etc.

In the next sections we will explore human-engineered features, namely Classical Features (Appendix A.1) and Random-Walk based features (Appendix A.2). Lastly, we will mention the limitations of machine learning approaches (Appendix A.3), which will show the need for graph deep learning (Section 2.1).

A.1 Classical Features

Classical features mainly revolve around centralities. They are distinguished into node,edge and graph-level features.

A.1.1 Classical Node Features

Possible node-level classical features are:

- Node degree
- Node Centralities
- Clustering Coefficient
- Graphlets

Node degree

It is the number of edge stubs connected to a node.

NB: Therefore, self-edges add 2 to the degree of node

Node Centralities

We will be reviewing a few centralities that may be used as features in downstream node-level tasks.

Eigenvector Centrality

The intuition behind the *eigenvector centrality* is that two nodes are equal if they are surrounded by equally important nodes.

If we call c_v the "importance" of a v, then we may define

$$c_v = \frac{1}{lambda} \sum_{u \in N(v)} c_u$$

where λ is a positive normalization factor and $N(v)$ is the neighborhood of v.

One may rewrite it as:

$$Ac = \lambda c$$

Where it is made clear that computing the eigenvector centrality is indeed an eigenvector problem.

Observation: The graph is undirected => A is symmetric => λ_{max} is positive and unique by the Perron-Frobenius theorem. Then c_{max} can be used as a centrality.

Closeness centrality

The intuition behind the *closeness centrality* is that a node is important if it has small shortest path lengths to all other nodes.

It is computed as:

$$c_v = \frac{1}{\sum\limits_{u \neq v} \text{length of shortest path between } u \text{ and } v}$$

Betweenness centrality

The intuition behind the *closeness centrality* node is that a node is important if it stands between many pairs of nodes.

It is given by:

$$c_v = \sum_{s \neq v \neq t} \frac{\text{n° of shortest paths between } s \text{ and } t \text{ that contain } v}{\text{n° of shortest paths between } s \text{ and } t}$$

Clustering Coefficient

It measures how many edges between the neighbors of a node v occur over how many would be possible. It ranges between 1 (clique) to 0 (no edges between neighbors).

It is given by (directed network):

$$e_v = \frac{\sum_{i,j \mid i,j \in N(v)} A_{ij}}{k_{in,i}(k_{in,i} - 1)}$$

Observation: Thus, The clustering coefficient counts the number of triangles in the ego-network of a node. Anyway, triangles are not the only structures a node may be part of. Moreover, complex structures may carry information as well. Therefore, we will call those structures *graphlets* (which comprise the individual link as an edge case) and proceed to model them into a centrality.

Graphlets

Definition A.1.1 *A graphlet is a rooted connected non-isomorphic subgraph.*

Observation: *Rooted* means that a graphlet is defined with respect to one of its nodes. Note that some nodes in a graphlet can be isomorphic, thus different choices of *root node* may actually lead to the same graphlet (which are then called *isomorphic* graphlets). See Figure A.1.

Definition A.1.2 *Having chosen a list of l graphlets, the graphlet degree vector (GDV) is a vector of length l associated to each node whose $i-th$ component counts the number of graphlets of the $i-th$ type within the list that the node is the root of.*

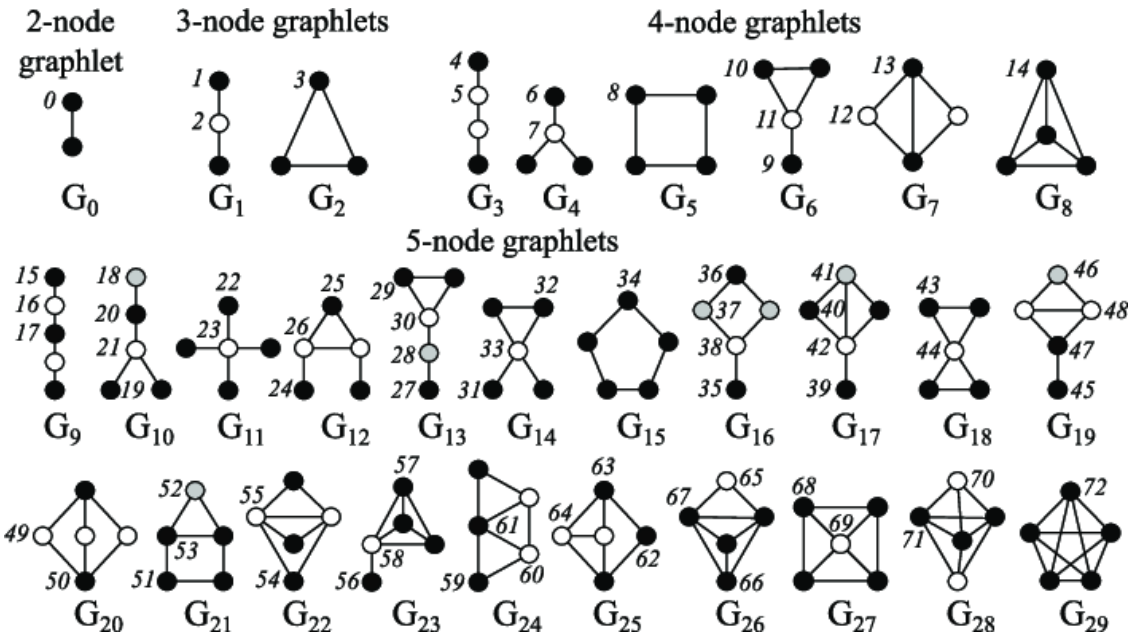


Figure A.1: Graphlets up to size 5. *Courtesy of Stanford CS224W*

Observation: The *GDV* gives more topological information than the simple degree vector. It is an extension of both the degree vector and of the clustering coefficients (which counts the occurrences of one specific graphlet, the triangle).

Summary

Importance-based features (e.g. centralities) are used to predict importances of other nodes, while structure-based features (e.g. clustering coefficient, graphlet degree vector) are useful to predict the role of a node in a network. The node degree can be seen as either an importance-based feature or a structure-based feature, since it is a degenerate case of both.

A.1.2 Classical Edge-level features

A common task over graphs is link prediction (Section 2.4.2). Briefly, link prediction consists in predicting missing links from a network.

The most common link prediction-tasks can be divided in:

1. *Links missing at random*: Some random links in the network are missing,

and an algorithm needs to find them. Useful for e.g. completing the protein interactome networks.

2. *Links over time*: Given the graph at time t_0, $G(t_0)$, we want to predict the *ranked list* L of links that will appear at time $t > t_0$. This is a more natural way of framing link prediction tasks for temporal networks e.g. transaction networks.

Such tasks may benefit from information related to pairs of nodes being encoded differently than just concatenating the node-level feature vectors.

Let us briefly review the most common edge-level classical features:

- Proximity
- Distance-based feature
- Local neighborhood overlap
- Global neighborhood overlap

Proximity

It consists in computing a score $c(x, y)$ defined as the number of common neighbors of x and y:

$$c(x, y) = |N(x) \cap N(y)|$$

Then sorting the pairs of nodes based on such score and finally selecting the top n as the candidate new links.

Distance-based feature

An example would be the length of the shortest path between the two nodes. Anyway, it has the disadvantage of not capturing neighborhood overlap or the number of paths. For that, we develop the next features.

Local Neighborhood Overlap

It is a set of features aimed at computing the number of neighbors that a pair of nodes has in common. here follow some examples of such features.

Cardinality of the intersection of neighborhoods

The most basic local neighbor overlap feature between two nodes u and v. It is computed as

$$|N(u) \cap N(v)|$$

Jaccard's Coefficient

A slightly more nuanced metric than the previous one, that normalizes the intersection of the thw neighborhoods by their union:

$$J(u, v) = \frac{N(u) \cap N(v)}{N(u) \cup N(v)}$$

Adamic-Adar Index

It is the sum of inverse logs of the degree of common neighbors. This way, pairs of nodes with lower-degree neighbors in common are computed as "more connected" than pairs of nodes with only celebrities in common. It is very suited for social networks. It is given by:

$$A_A(u, v) = \sum_{x \in N(u) \cap N(v)} \frac{1}{\ln(k_u)}$$

Global Neighborhood Overlap

Observation: Local Neighbor Overlap metrics return zero if the two nodes do not have a neighbor in common. But this shouldn't prevent the two nodes from being connected in the future. Thus, we define *Global Neighborhood Overlap* metrics, such as the *Katz Centrality*.

Katz Centrality

The intuition is to count all paths of a certain length between a pair of nodes, and sum these counts each weighted by the respective length. This should give an estimation of "how much connected" the two nodes are. The formula for the Katz index S_{uv} between two nodes u and v reads:

$$S_{uv} := \sum_{l=1}^{+\infty} \beta^l * \text{n}^\circ \text{ of paths between } u \text{ and } v \text{ of length } l = \sum_{l=1}^{+\infty} \beta^l * A_{uv}^l$$

The second equality holds since:

1. A_{uv} is the number of paths between u and v of length 1;
2. $\sum_i A_{ui}Aiv = A^2_{uv}$ is the number of paths between u and v of length 2;
3. $\sum_i A^2_{ui}A_{iv} = A^3_{uv}$ is the number of paths between u and v of length 3;

l. A^l_{uv} is the number of paths between u and v of length l.

Therefore, recognizing the geometric series, in matrix form we have:

$$S = \sum_{l=1}^{+\infty} \beta^l A^l = (\mathbb{I} - \beta A)^{-1} - \mathbb{I}$$

An extension of the Katz centrality to temporal networks may be found in [53].

A.1.3 Classical Graph-level Features

We are going to briefly review the perhaps most well known methods for graph embedding, namely kernel-based methods.

Kernel-based methods for Graph Embedding

Definition A.1.3 *A kernel is a positive semidefinite (hence symmetric) linear map K .*

Observation: Once we will have defined the vector embeddings for two graphs G and G', we will be able to compute $K(G, G')$ i.e. the kernel-induced similarity between the two graphs. These kernels may be used e.g. in SVMs.

Observation: Given a kernel K, there always exist a vectorial *feature representation* ϕ of K such that:

$$K(G, G') = \phi(G)^T \phi(G')$$

.

The two graph kernels we are going to discuss are:

- Graphlet Kernel
- Weisfeiler-Leman Kernel

Graphlet Kernel

It consists in defining a list of k graphlets:

$$\mathcal{G} = (g_1, g_2, ..., g_k)$$

So that we may later count the number of times each graphlet appears in the graph G:

$$\phi_{graphlet}(G) := (n_1, n_2, ..., n_k)$$

Hence the kernel.

Observation: Graphlets are by convention taken as non-rooted in this context.

Observation: Since a difference in the size of the two graphs may play as a confounding factor, it is best to scale the feature representation by its sum:

$$\tilde{\phi}_{graphlet}(G) := \frac{\phi_{graphlet}(G)}{\sum_i \phi_{graphlet}(G)_i}$$

Observation: This method presents computational issues. In fact, the complexity counting graphlets is exponential in the graphlet size. Therefore we present the following, more efficient method.

Weisfeiler-Leman kernel

Central to the Weisfeiler-Leman algorithm is the concept of multiset that we presently describe.

Definition A.1.4 *A multiset $\mathcal{S}$ is a couple $\mathcal{S} = (S, m)$ where S is a set and $m : S \to \mathbb{N}$. Intuitively, a multiset is an unordered collection with repeat elements, and m counts the number of times each element is repeated. We will denote multisets using slashed curly brackets $\{\!\{\}\!\}$*

In order to introduce the Weisfeiler-Leman kernel, we first need to define graph isomorphism [6].

Definition A.1.5 *Given two graph $G = (V, E)$ and $G' = (V', E')$, we say that they are isomorphic $\iff$ $\exists$ bijection $g : V \to V' | (u, v) \in V \implies (g(u), g(v)) \in V'$*

Algorithm 1 One iteration of the 1-dim. Weisfeiler-Lehman test of graph isomorphism

1: Multiset-label determination
- For $i = 0$, set $M_i(v) := l_0(v) = \ell(v)$. [2]
- For $i > 0$, assign a multiset-label $M_i(v)$ to each node v in G and G' which consists of the multiset $\{l_{i-1}(u) | u \in \mathcal{N}(v)\}$.

2: Sorting each multiset
- Sort elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$.
- Add $l_{i-1}(v)$ as a prefix to $s_i(v)$ and call the resulting string $s_i(v)$.

3: Label compression
- Sort all of the strings $s_i(v)$ for all v from G and G' in ascending order.
- Map each string $s_i(v)$ to a new compressed label, using a function $f : \Sigma^* \to \Sigma$ such that $f(s_i(v)) = f(s_i(w))$ if and only if $s_i(v) = s_i(w)$.

4: Relabeling
- Set $l_i(v) := f(s_i(v))$ for all nodes in G and G'.

Figure A.2: Weisfeiler-Leman test. i denotes the iteration, $M_i(v)$ the multiset label to assign to node v at iteration i, $l_0(v)$ is the initial label of node v, which could be the same number $\forall v$ or its degree, sorting is lexicographic. *Courtesy of [6]*

Observation: Given two graphs $G_1 = (V, E)$ and $G' = (V', E')$, it is very expensive to test for their isomorphism by algorithmically looking for such an g. Moreover, its complexity is not known (see Wikipedia). Therefore, approximate algorithms have been developed, the most famous of which is the Weisfeiler-Leman test. It consists in the algorithm in Figure A.2.

It will terminate after the fourth step of iteration $i \iff$:

$$\{\!\{l_i(v) | v \in V\}\!\} \neq \{\!\{l_i(v) | v \in V'\}\!\} \tag{A.1}$$

And will deem the two graphs non-isomorphic. Otherwise, it will terminate after 'maxiter'. The f function in the algorithm is an *hash* function, that for us will suffice to be injective. Note that being the Weisfeiler-Leman kernel approximate, when it fails the two graphs are guaranteed to be non-isomorphic, but when it reaches 'maxiter' we cannot conclude that they are.

Drawing its ideas from bag-of-words approaches to text embedding, the kernel associate to the Weisfeiler-Leman test, known as *Weisfeiler-Leman (subtree) kernel*, requires to compute the following feature representation [6]:

1. Assign an initial color $c_0(v)$ to each node v (e.g. its degree), and choose a number of iterations K;

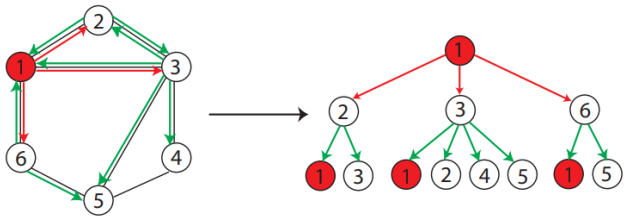


Figure A.3: A subtree pattern of height 2 rooted at the node 1. Note the repetition of nodes in the unfolded subtree pattern on the right. *Courtesy of [6]*

2. At each iteration i, for each node v, define the *multiset* $(S_{i,v}, m)$ where $S_{i,v} = \{c_k(v)\} \cup \{c_k(u) \mid u \in N(v)\}$ and $m : S_{i,v} \to \mathbb{N}$ counts the number of times each element of $S_{i,v}$ was present before aggregation. Assign to each node the color given by an hashing of $S_{i,v}$, $c_{k+1}(v) = HASH(S_{i,v})$;
3. Repeat 2. for K times. Each repetition is called *color refinement*. At that point, each node will have encoded the structure of its K-hop neighbors.

Observation: Due to the recursive manner in which the Weisfeiler-Leman algorithm aggregates the colors of a node's neighbors, K-iterations of it over the same node v can be represented by a *tree subgraph rooted* in v, where colors are aggregated from the bottom up. See Figure A.3 for a pictorial representation.

Observation: After repeating the color refinement K times, a feature representation is defined such that it counts the number of times each color appeared taking into account every color refinement iteration. If we define Σ as the set of all colors that appeared during the application of the Weisfeiler-Leman kernel in both G and G', then the graph-level feature is:

$$\phi_{Weisfeiler-Leman}(G) = \vec{p}(\Sigma|G)$$

Where $\tilde{p}(\Sigma|G)$ is the empirical distribution of the elements of Σ in G throughout all iterations. Please refer to Figure A.4 for an illustration.

Observation: The $HASH$ function (by definition) is injective over the multiset. An easy way to obtain such function would be to sort the multiset into an ordered sequence and map each unique sequence to a different natural

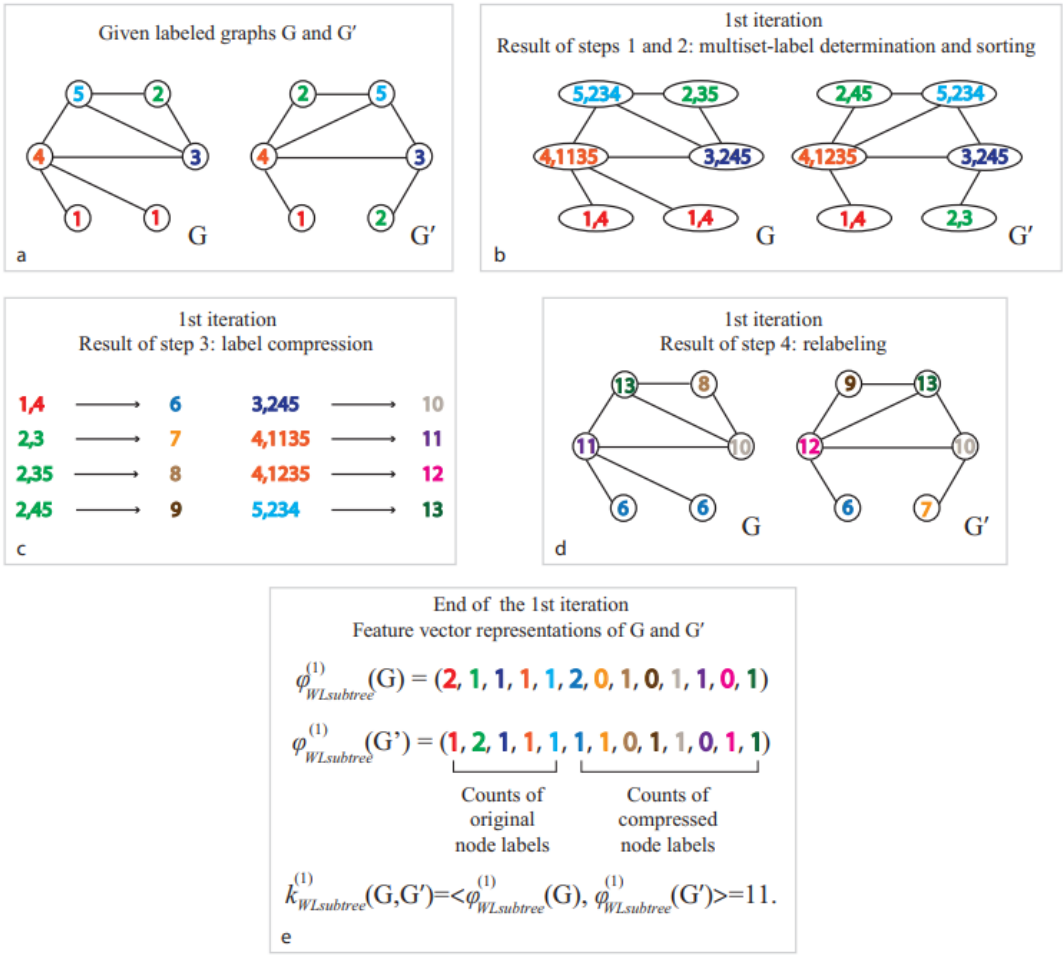


Figure A.4: Illustration of the computation of the Weisfeiler-Leman subtree kernel with $h = 1$ for two graphs. Here $\{1,2,...,13\} \in E$ are considered as letters. Note that compressed labels denote subtree patterns: For instance, if a node has label 8, this means that there is a subtree pattern of height 1 rooted at this node, where the root has label 2 and its neighbors have labels 3 and 5. *Courtesy of [6]*

number. Due to the injectivity of the $HASH$, A Weisfeiler-Leman test with K iterations may distinguish neighborhoods that present structural differences at the K-th order at most.

A.2 Random Walk-based Embedding Techniques

A slightly less data-agnostic way of computing features requires using random walks. Intuitively a random walk over a graph is a discrete process where at every iteration a new node is selected according to some rules. Given a node v, random walks induce a neighborhood $N_{RW}(v)$ of v defined by all nodes visited by an instance of the process that started from v. By specifying an embedding similarity function, random walks therefore allow to optimize embeddings so that embeddings of nodes belonging to $N_{RW}(v)$ are "close" to the embedding of v. This amounts to a shallow, unsupervised approach to node embedding. The resulting embeddings won't be task-specific, but at least they'll be network-specific.

We are going to describe random walk-based approaches to node and graph embeddings, and their extensions to temporal graphs.

A.2.1 Node Embedding: General Framework

More concretely, I've so far interpreted node embedding of $G = (V, E)$ as determining a function f s.t. :

$$f : V \to \mathbb{R}^d$$

Now, given a graph-level measure of node similarity

$$\text{N-similarity} : V \times V \to \mathbb{R}$$

Which will for example be the shared membership to the same RW-induced neighborhood, we also introduce an embedding level similarity:

$$\text{E-similarity} : \mathbb{R}^d \times \mathbb{R}^d$$

An example of such E-similarity could be:

$$\text{E-similarity}(f(u), f(v)) := f(u)^T f(v)$$

To get an unsupervised loss, we require that:

$$\text{N-similarity}(u, v) \approx \text{E-similarity}(f(u), f(v))$$

Therefore, we could for example minimize:

$$\mathcal{L} = \sum_{u,v \in V} \text{N-similarity}(u, v) - f(u)^T f(v)$$

Observation: Different node embedding algorithms will specify different N-similarities, different losses and different loss optimization algorithms.

Observation: Embeddings of this sort do not make use of node labels.

Observation: Embeddings of this sort are task independent.

A.2.2 Random Walk Approaches to Node Embedding

Definition A.2.1 *A random walk (or random process) is a process on a graph where a node is selected, one of its neighbor is randomly chosen and the same is repeated for a fixed number of times.*

Definition A.2.2 *At each iteration of the RW process, a node is randomly selected from a set. The rule by which the neighborhood of a node is defined and a random neighbor is selected is known as random walk strategy or RW-strategy and will be abbreviated by R.*

Observation: Note that R completely specifies the RW process, so we will use R to denote it.

Notation: Given a random walk R, we indicate the neighborhood of a node $u \in V$ *as defined by a radom process R that starts in u* via $N_R(u)$. v's membership to $N_R(u)$ will denote N-similarity between u and v.

Notation: Given a random walk R, we indicate with $P(f(v)|N_R(u))$ the probability that the node v, as represented by the embedding f, is part of the R-induced neighborhood $N_R(u)$ of u. This will be the E-similarity.

Observation: The task of the optimization algorithm will be to find f such that when two nodes are N-similar, then they're also E-similar:

$$\mathcal{L} := \sum u \in V \sum_{v \in N_R(u)} \ln(P(f(v)|N_R(u)))$$

is maximized.

We are now going to see two examples of RW-based node embedding algorithms:

- DeepWalk
- Node2Vec

Which will specify R and $P(f(v)|N_R(u))$.

DeepWalk

The *DeepWalk* (DW) algorithm specifies:

- R is a random walk with uniform probability, and will be denoted as DW. Moreover, for every node u, a fixed number of random walks is run and all the visited nodes are considered to be part of $N_{DW}(u)$;
- $$P(f(v)|N_{DW}(u)) := \frac{e^{f(u)^t f(v)}}{\sum\limits_{n \in V} f(u)^t f(n)} \tag{A.2}$$

Which is the softmax function. Maximizing $\mathcal{L}$ with respect to f thus requires moving as much conditional probability mass as possible to the neighborhood of each node.

Observation: $\mathcal{L}$ is given by two nested sums $\implies O(|V|^2)$ complexity. One may approximate $\mathcal{L}$ with *negative sampling* i.e.:

$$\ln\left(\frac{e^{f(u)^t f(v)}}{\sum\limits_{n \in V} f(u)^t f(n)}\right) \approx \ln(\sigma(f(u)^t f(v))) - \sum_{i=1}^{k} \ln(f(u)^t f(n_i))$$

where σ is the sigmoid function and the n_i are extracted from a distribution P_V. Usually $P_V(v) \sim deg(v)$. k is usually between 5 and 20.

This approach changes the loss, but can be shown to well-approximate the original one.

Observation: Usually, the algorithm chosen to extremize the loss is the stochastic gradient descent.

Observation: It is possible to get better embeddings than DeepWalk's by employing another algorithm, Node2Vec.

Node2Vec

Node2Vec uses the same embedding similarity as DeepWalk's Equation (A.2), but modifies R so that whenever the RW process transits from a node u to a node v, it may (all probabilities are unnormalized):

1. Go back to u with probability $\frac{1}{p}$;
2. Transit to a node within 1 hop with probability 1;
3. Jump to a more distant node with probability $\frac{1}{q}$.

The interpretation is the following: p regulates the breadth first search-like behavior, while q the depth first search-like behavior.

Observation: Node2Vec is generally more efficient then DeepWalk, and produces better embeddings (i.e. higher performances on downstream tasks).

A.2.3 Random Walk Approaches to (Sub)Graph Embedding

In this section we are going to list the most popular random walk-based methods to (sub)graph embedding, namely:

- Averaging over RW-based node embeddings;
- Virtual Nodes
- Anonymous Walks

Later, we will also describe an extension to Anonymous Walks that comprises temporal networks.

Averaging over RW-based node embeddings

This technique consists in obtaining the embedding of a (sub)graph by averaging over the embeddings of the nodes it comprises, that may have been obtained e.g. via Node2Vec.

Virtual Nodes

This technique consists in introducing a *virtual node* i.e. a node that is connected to all the nodes of the (sub)graph we wish to embed, and then running a node embedding algorithm e.g. Node2Vec over the entire graph. The embedding of the virtual node should then be taken as the embedding of the (sub)graph of interest.

Anonymous Walks

This technique consists in running m random walk processes of length l over the graph, extracting the subgraphs that these processes highlight on the network, and then for each subgraph anonymize the nodes by relabelling them using the first time step they were visited in by the RW process. Each anonymized subgraph is an *anonymous walk* (AW). See Figure A.5 for an illustration.

Note that the number η of non-isomorphic anonymous walks of length l increases with l. Moreover, the number m of anonymous walks of length l required to embed a graph with probability less than δ to have an error on the distribution of AWs greater than ϵ is:

$$m = \left[\frac{2}{\epsilon^2}(\ln(2^\eta - 2) - \ln(\delta))\right]$$

The graph embedding will then be given by the empirical distribution of the sampled anonymous walks.

Embedding of Temporal Graphs via Causal Anonymous Walks

It has been proven that the evolution of temporal graphs usually reflects established temporal network motifs [51].

An important drawback of anonymous walks is that they are anonymized based on the single walk and thus loose correlations between network motifs.

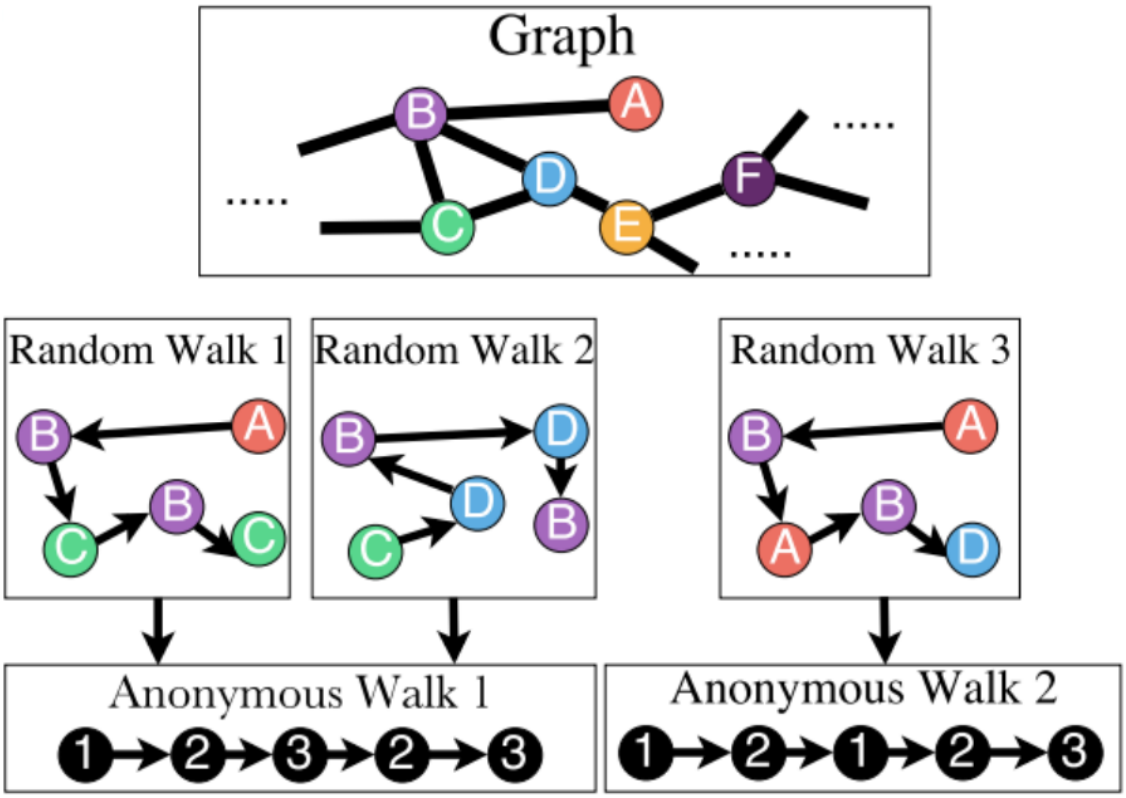


Figure A.5: Anonymous Walks. *Courtesy of Stanford CS224W*

A solution to this problem was proposed in [51], where *causal anonymous walks* (*CAW*s) are proposed i.e. an extension of anonymous walks to temporal graphs.

A temporal network $G = (V, E)$ contains links that are available only after a certain time. Thus, the general element of E will be denoted by (e, t) where t is the time when edge e is made available. An *availability-preserving* walk on such networks is then given by:

$$W = ((w_0, t_0), (w_1, t_1), ..., (w_m, t_m))$$

where $t_0 > ... > t_m$ (so it is given backward) and the w_is are the nodes. We will denote by $W[i][0]$ and $W[i][1]$ respectively the node and the time stamp of the i-th transition. The AW-embedding of a node w in an random walk W, according to section A.2.3, would be given by:

$$I_{AW}(w; W) := |\{v_0, v_1, ..., v_{k_*}\}|$$

where k_* is the smallest index s.t. $v_{k_*} = w$

Such node embedding only reflects the positions of the node within the random walk and doesn't take into account correlations between different random walks, which are implied by the temporal motifs we know to drive the network's evolution.

To make up for it, the authors propose to define S_{w_0} as the set of all random walks starting from w_0, and embed a node w with respect to such set via:

$$g(w, S_{w_0})[i] := |\{W | W \in S_{w_0} \wedge w = W[i][0]\}|$$

If the task we are preparing for is link prediction, we may define the concatenated embedding:

$$I_{CAW}(w; \{S_u, S_v\}) := \{g(w, S_u), g(w, S_v)\}$$

The embedding of a walk is finally given by:

$$\hat{W} = ((I_{CAW}(w_0, \{S_u, S_v\}), t_0), (I_{CAW}(w_1, \{S_u, S_v\}), t_1), ..., (I_{CAW}(w_m, \{S_u, S_v\}), t_m))$$

The authors then proceed to encode such embeddings using RNNs. That is, every random walk $\hat{W}$ is encoded by the final internal state $\vec{h}_m$ of an RNN that at each iteration tries to predict the next CAW-embedded node:

$$enc(\hat{W}) = RNN(\{f_1(I_{CAW}(w_i; \{S_u, S_v\})) \bigoplus f_2(t_{i-1} - t_i)\}_{i=0,..,m+1}) \text{and} t_{-1} = t_0$$

Refer to [51] for details on f_1 and f_2.

After each CAW-embedded random walk has been encoded, the final encoding of $S_u \cup S_v$ is obtained by averaging all the encodings of the $\hat{W}$ they contain, or via a self-attention mechanism. A final 2-layer MLP over the encodings of $\{S_u \cup S_v\}$ is used to make the final link prediction between u and v.

The authors prove the superiority of their embeddings with respect to the state of the art in subsequent inductive and transductive link prediction tasks.

A.2.4 PageRank

PageRank is a node ranking algorithm used, in some of its variants, by Google Search. Here it is reported under the section related to "Random Walks" because of its ample connections to such concept that we shall explore later.

The idea behind PageRank is that a node must be "more important" if it receives inward links (in-links) from other important nodes. Each in-link is in fact viewed as a "vote" from the node that emanates it, and carries with it an importance equal to the importance of its source node divided by its out-degree.

Thus, the importance r_v of node v is given by:

$$r_v = \sum_{u \to v} \frac{r_u}{d_u}$$

Where d_u is the out-degree of node u.

We will use the following definition:

Definition A.2.3 *Given a graph G, we define its stochastic adjacency matrix M as:*

$$M_{uv} := \frac{1}{d_v}$$

Observation: M is column-stochastic i.e. the sum of each of its columns is 1.

If we furthermore enforce that

$$\sum_u r_u = 1$$

We get a condition for $\vec{r}$:

$$\vec{r} = M\vec{r} \tag{A.3}$$

Which is a linear system for $\vec{r}$.

Connection to Random Walks

Given an RW process on the graph, if $p(\vec{t})$ is the vector of probabilities of finding the process in each node $v \in V$, and if the RW strategy requires to chose the next node from the topological neighbors of the present node (like DeepWalk), then we have:

$$p(t \vec{+} 1) = Mp(\vec{t})$$

Thus we see that the Equation (A.3) describes the stationary distribution of such a process, hence the connection between PageRank and Random Walks.

Observation: Solving for the linear system is a method that doesn't scale well with network size. Anyway, we note that Equation (A.3) may also be interpreted as the eigenvector equation for M with eigenvalue 1. Turns out (see Wikipedia) that 1 is the highest eigenvalue of M, thus $\vec{r}$ is also the *principal eigenvector* of $M \implies$ we can use the *Power Iteration method* (*PI-method*).

The method consists in choosing an initial guess for the rank vector:

$$\vec{r}^{(0)} = (\frac{1}{N}, .., \frac{1}{N})^T$$

and a positive ϵ. Then one iterates:

$$r_v^{(t+1)} = \sum_{u \to v} \frac{r_u^{(t)}}{d_u}$$

Until

$$|\vec{r}^{(t+1)} - \vec{r}^{(t)}|_1 < \epsilon$$

Corrections to PageRank

Observation: The PI-method converges, but may produce results negatively influenced by:

1. *Dead ends*, i.e. nodes with 0 out-degree;
2. *Spider traps*, i.e. clusters with low overall out-degree.

The formers break the required column stochasticity of M, invalidating the procedure, while the latters absorb all the importance.

Therefore, it is necessary to mitigate such inconveniences by modifying the method or the graph.

The solution to spider traps is to add a probability β to, at each iteration, random jump to a node in the network that is not in the neighborhood of the present node. This leads to a modification of the PageRank equation.

The solution to dead ends is to connect them to every other node in the network, and run the RW process normally.

After modifying the graph to account for dead ends, the solution for spider traps implies a change in the PageRank's equations:

$$r_v = \sum_{u \to v} \beta \frac{r_u}{d_u} + (1 - \beta)\frac{1}{N}$$

It reads as: given a node v, its importance is given by the importance of each of its in-neighbors divided by their out-degree times the probability to visit one of them plus (and the next term is outside the sum) the probability to random-jump and land in u which is $(1 - \beta)$ times the inverse of the number of nodes. This formula assumes you have removed all dead ends like before.

If we define the *Google Matrix*:

$$G := \beta M + (1 - \beta) \left[\frac{1}{N}\right]_{N \times N}$$

Then we get the standard form back:

$$\vec{r} = G\vec{r}$$

And the PI-method would work again.

Personalized PageRank and Random Walks with Restarts

Given an user-item bipartite graph, one might be willing to implement a recommender system based on collaborative filtering] upon it. The idea is that, if an item P is usually bought together with an item Q, I'd like to recommend Q every time P is bought.

But we have to mine those similarities from the graph itself. The two metrics that we might want to look at are the number of neighbors each pair has in common (i.e. users who bought both items) and the shortest path between its two elements.

A good candidate feature to take into account both metrics is the PageRank algorithm where we modify the random jump mechanism to just occur towards a restricted number of elements i.e. those elements (or that element) we wish to compute the similarities of the others against.

We call the so-restricted set of elements S, the *teleport set*.

If $|S| = 1$, the PageRank is called *Random Walk with Restarts*, otherwise it is named *Personalized PageRank*

Matrix Factorization and RW-based Node Embeddings

We may define the N-similarity between nodes u and v using the adjacency matrix:

$$\text{N-similarity}(u, v) = A_{uv}$$

Thus when we look for an embedding $\vec{f}$ s.t. :

$$A_{u,v} \approx \vec{f}(u)^T \vec{f}(v)$$

We are effectively trying to factorize A.

Observation: Since the dimensionality of the embedding d is much smaller than the number of rows N of A, such factorization may only be approximate, thus we want to minimize the loss:

$$||A - F^T F||_f$$

where $||.||_f$ is the Frobenius norm, and F is the embedding matrix that has the vectors $\vec{f}(v)$ as columns.

Observation: DeepWalk and Node2Vec can be cast into a matrix factorization problem, where the matrix to factorize, in case of DeepWalk, is given by:

$$\ln(vol(G)\left(\frac{1}{T}\sum_{r=1}^{T}(D^{-1}A)^r\right)D^{-1}) - \ln b$$

Observation: Node2Vec would have a more complex matrix, but the concept is the same.

Thus, *PageRank is equivalent to computing the stationary distribution of a Random Walk, and RW-based node embedding methods can be reduced to matrix factorization problems.*

A.3 Limitations of Graph Machine Learning models

The main limitations of classical and RW-based approaches are:

1. Features are *human-engineered*, which requires considerable effort on top of the subsequent classifier/regressor model selection, tuning and training;
2. Computed features are not task nor model-specific, so they could unpredictably perform better at one task but worse at another. Performance swings could also be observed by changing model;

3. These approaches cannot use node, edge nor graph metadata;
4. Since features computation and model training are two separate steps, these approaches are not *end-to-end*;

Moreover, RW-based embeddings suffer from:

1. Inherent *transductiveness*, meaning that to obtain embeddings for new nodes that are not in the training set, or nodes of a completely new graph, one has to recompute embeddings for all nodes. This is very limiting for ML applications, where new nodes could appear at later times;
2. Impossibility to capture structural similarities. Given two node clusters structurally very similar, Deep-Walk or Node2Vec will come up with different embeddings since they look at node identities. On the other hand, Anonymous Walks may do better, since they forget node identities;

Similar issues are shared by machine learning models across many modalities, not only graphs. Therefore, Deep Learning architectures were developed over the years, which are capable of addressing all of the above concerns simultaneously.

Appendix B

Performance-Oriented Architectures and Techniques

B.1 (Advanced) Cluster GCNs

Besides Neighbor Sampling, another approach to computational issues is that of *Cluster GCNs* (*CGCNs*). These are standard GCNs with an added preprocessing step where the network is partitioned into subgraphs small enough to fit into the GPU memory, and then full-batch training is performed on each of them.

The first observation is on how to partition the graph into subgraphs. Since we want our gradients to be representative of what I'd have with a full batch, each subgraph should preserve as many edges of its original nodes as possible. For this reason, the partitioning is usually performed with the Louvain or BigCLAM algorithms, which, by choosing subgraphs of well-connected nodes, only sever the so-called *long edges*.

The drawback of this approach is that the long edges are of vital importance for some downstream tasks, since they allow the information to reach distant parts of the network. Also, by training on batches of similar nodes, the gradients of CGCNs tend to have high variance and are thus unreliable. For these reason, *Advanced Cluster GCNs* (*ACGCNs*) were invented. Similarly to

CGCNs, these GNNs also perform community-based clustering in their pre-processing step, but they tune the partitioning algorithms so that they find clusters small enough that aggregating some of them is still computationally tractable in the GPU. Later, clusters are randomly selected and aggregated together to perform full batch training on the resulting subgraph. The aggregation of randomly-chosen small clusters allows for some long edges to be retained, and this usually improves and performance.

B.2 Simplified GCNs

A completely different perspective on improving the performance of GNNs has been undertaken via *Simplified GCNs* (*SGCNs*) ([34]). Instead of sampling neighbors or adding a pre-processing clustering step, SGCNs lighten the computational cost by removing their nonlinearities.

That is, a standard GNN layer looks like:

$$\overrightarrow{h}_v^{(k+1)} = \sigma\left(W_{k+1} \sum_{u \in N(v)} \frac{\overrightarrow{h}_u^{(k)}}{|N(v)|}\right)$$

Thus:

$$H^{(k+1)} = \sigma(D^{-1}AH^{(k)}W_{k+1}^T) = \sigma(\tilde{A}H^{(k)}W_{k+1}^T)$$

While its SGCN version would be:

$$H(k+1) = \tilde{A}H^{(k)}W_{k+1}^T$$

By iterative substitution we get, for the final K-th embedding:

$$H^{(K)} = \tilde{A}H^{(K-1)}W_K^T = \tilde{A}(\tilde{A}H^{(K-2)}W_{K-1}^T)W_K^T = ... = \tilde{A}^K \underbrace{X}_{\equiv H^{(0)}} \underbrace{\prod_{k=0}^{K} W_k^T}_{=:W} = \tilde{A}^K X W^T$$

So we can get the final embeddings from the initial feature matrix X with just one matrix operation. Thus, if we have N nodes, the complexity of SGCNs scales linearly with N.

We expect SGCNs to perform significantly worse on downstream tasks, since they defeat the whole purpose of Deep Learning, namely approximating complex functions by combining linearities with nonlinearities. And while this expectations is sometimes met, it has been shown that on real world networks SGCNs do not fare much worse than standard GCNs.

This happens because sometimes the network's connectivity reflects SGCNs' aggregation strategy. That is, real world network often exhibit *assortativity* i.e. the tendency of nodes with similar features to cluster together. On the other hand, SGCNs' nodes embeddings are produced via iterated averages over nodes' neighbors. This makes SGCNs' embeddings representative when assortativity is present, which is often the case in real-world networks.

See [34] for more details and experiments.

B.3 Comparison of Performance-Oriented Architectures and Techniques

Now Let us discuss how these methods relate to each other.

First of all, given M nodes to compute the embedding of using a K-layer GNN, the complexity of Neighbor Sampling with H nodes sampled per layer is MH^K (since one has to compute MH^K embeddings). Analogously, given a cluster with average degree D_{avg}, the complexity of a full batch pass of a ACGCN would be KMD_{avg}. Assuming $H = \frac{D_{avg}}{2}$ (usually H is more than that), ACGCNs are still more efficient than Neighbor Sampling, although learning can be unstable.

SGCNs, on the other side, do not need a clustering pre-processing step and thus enjoy a more stable learning, but suffer from inferior performance when the assortativity condition does not hold.