Bachelor's Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Graph Neural Networks and their Temporal Variants

Jiří Liška

Supervisor: Ing. Radek Mařík, CSc.
Field of study: Open Informatics
Subfield: Artificial Intelligence and Computer Science
May 2024

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Liška Jiří**	Personal ID number:	**498838**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Cybernetics**		
Study program:	**Open Informatics**		
Specialisation:	**Artificial Intelligence and Computer Science**		

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Graph Neural Networks and Their Temporal Variants

Bachelor’s thesis title in Czech:

Grafové neuronové sítě a jejich temporální varianty

Guidelines:

The aim of the project is to create an overview of methods dealing with graph neural networks and their temporal modifications. The review will focus on their properties and possibilities of application deployment.
Steps:
1. Create a survey of methods focusing on graph neural networks (GNNs). Include also their temporal modifications (tGNNs).
2. Select an appropriate dataset to demonstrate the specific properties of GNNs and tGNNs.
3. Implement a suitable experimental environment.
4. Verify experimentally the selected GNN and tGNN properties and provide a discussion of the obtained results.

Bibliography / sources:

[1] Lingfei Wu, Peng Cui , Jian Pei, and Liang Zhao: Graph Neural Networks: Foundations, Frontiers, and Applications. Springer, 2022.
[2] Zhiyuan Liu, and Jie Zhou: Introduction to Graph Neural Networks, Morgan & Claypool, 2020.
[3]. Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun: Graph neural networks: A review of methods and applications, AI Open 1, 57–81, 2020.

Name and workplace of bachelor’s thesis supervisor:

Ing. Radek Mařík, CSc. Department of Telecommunications Engineering FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **15.01.2024** Deadline for bachelor thesis submission: **24.05.2024**

Assignment valid until: **21.09.2025**

Ing. Radek Mařík, CSc. Supervisor’s signature	prof. Dr. Ing. Jan Kybic Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt Student’s signature

Acknowledgements

I would like to sincerely thank my supervisor, Ing. Radek Mařík, CSc., for his advice and continual mentorship.

I am very grateful to my family for their persistent support throughout my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodological instructions for observing ethical principles in the preparation of university theses.

In Prague, May 23, 2024.

. . . .

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 23.května 2024.

. . . .

Abstract

This thesis aims to provide an overview of Graph Neural Networks (GNNs), a branch of neural networks designed to work with graph data. Graphs are a general data structure that can be used to represent a wide range of complex domains, including social networks, molecule interactions and knowledge graphs. GNNs have surpassed performance of traditional machine learning algorithms and handcrafted heuristics on many graph datasets.

Until recently, graph neural networks have been mainly applied to static graphs. However, many real-world applications involve dynamic graphs, where the underlying graph structure changes over time. Extending graph neural networks to dynamic graphs has proven to be a challenging task. Nevertheless, temporal variants of graph neural networks have shown promising results on real-world graph datasets. Both static and temporal graph neural networks are reviewed in the thesis.

Experiments with both static and dynamic graphs were performed and evaluated. Static graph neural networks have been applied to a novel domain of hieroglyph classification and achieved state-of-the-art performance compared to previously used methods.

Keywords: Graph Neural Networks, GNN, Temporal Graph Neural Networks, message passing, node classification, dynamic graph

Supervisor: Ing. Radek Mařík, CSc.

Abstrakt

Cílem této práce je poskytnout přehled grafových neuronových sítí (GNN), které jsou odvětvím neuronových sítí navrženým pro práci s grafovými daty. Grafy jsou obecnou datovou strukturou, která může být použita k reprezentaci široké škály komplexních domén, včetně sociálních sítí, interakcí molekul a znalostních grafů. Grafové neuronové sítě překonaly výsledky tradičních metod strojového učení a ručně vytvářených heuristik na mnoha sadách grafových dat.

Donedávna byly grafové neuronové sítě aplikovány především na statické grafy. Nicméně mnoho reálných aplikací zahrnuje dynamické grafy, kde se základní struktura grafu mění v čase. Rozšíření grafových neuronových sítí na dynamické grafy se ukázalo jako náročný úkol. I přesto však temporální varianty grafových neuronových sítí ukázaly slibné výsledky na reálných grafových sadách data. Tato práce shrnuje a popisuje statické i temporální grafových neuronové sítě.

Byly provedeny experimenty se statickými i dynamickými grafy, včetně jejich vyhodnocení. Grafové neuronové sítě byly použity na novou úlohu klasifikace hieroglyfů a překonaly výsledky dříve používaných metod.

Klíčová slova: Grafové neuronové sítě, GNN, Temporální grafové neuronové sítě, předávání zpráv, klasifikace uzlů, dynamický graf

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Motivation

Graphs are a ubiquitous data structure. Many real-world systems can be modeled as graphs, for example, computer networks, social networks, transportation networks, and others. Graphs have been extensively researched in mathematics, but up until recently, they have not been the main focus of the machine learning community.

This thesis aims to provide an overview of Graph Neural Networks (GNNs), a class of neural networks designed to work with graph data. Graph neural networks have been successfully used in areas such as recommendation systems, knowledge graphs, network analysis (to forecast future states), chemistry (predicting molecule structure), code analysis, and others (Wu et al., 2022).

Due to the interconnected nature of graph data, computations in GNNs require more care than traditional machine learning methods when scaling to large datasets. Efficient sampling methods for scaling GNNs to graphs with tens of millions of nodes were discovered in recent years and are discussed in the text as well.

Although analysis of static graphs can provide valuable insights, graphs in many areas are changing in time and are thus dynamic by nature. Variants of graph neural networks, called temporal graph neural networks, were proposed to handle dynamic graphs. The thesis will discuss selected significant state-of-the-art temporal graph neural networks and compare them to static graph neural networks.

1.2 Structure

Structure of the thesis is provided below, to facilitate easier navigation for the readers. In Chapter 2 the general concepts and terminology of graphs and learning methods are introduced. Following them is the review of static graph neural networks. Dynamic graphs, along with corresponding temporal graph neural networks are introduced in Chapter 3. Chapter 4 discusses the available tools for implementations of graph neural networks. Design of the experiments and details about used datasets and methods are provided in Chapter 5. In Chapter 6 the results of the experiments are presented and discussed.

Last two chapters are dedicated to considerations of future work, in Chapter 7, and overall conclusion of the thesis in Chapter 8.

Chapter 2

Graph Neural Networks Review

The definitions in the following sections are derived from the book (Wu et al., 2022).

2.1 Terminology

Definition 2.1 (Graph)**.** Graph $G = (V, E, X, X_e)$, where V is the set of nodes, E is a set of edges, X is an optional node feature matrix $X \in \mathbb{R}^{|V| \times d}$, where d is the number of node features. An edge feature matrix is also optionally included, denoted as $X_e \in \mathbb{R}^{|E| \times d_e}$, where d_e is the number of edge features.

An adjacency matrix of an undirected graph is a matrix $A \in \mathbb{R}^{|V| \times |V|}$ such that $A_{ij} = 1$ if there is an edge between nodes i and j, and 0 otherwise. Node degree d_i is the number of edges incident to node i.

2.2 Learning tasks

The most commonly used tasks in graph learning are described below.

Definition 2.2 (Node classification)**.** Given a graph G with a set of nodes V, node classification is a function

$$f_{NC} : V \to C$$

that maps nodes to a set of possible node classes C.

Definition 2.3 (Edge prediction (link prediction))**.** Given a graph G with a set of nodes V, edge prediction (link prediction) is a function

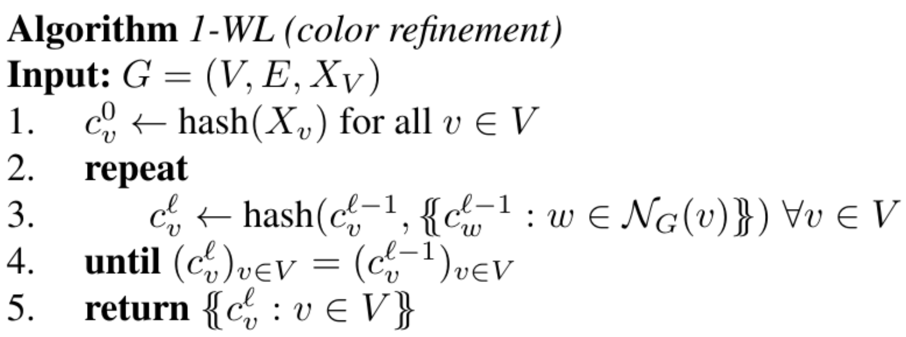
$$f_{EP} : V \times V \to [0, 1]$$

that predicts a probability that an edge between two nodes exists.

Definition 2.4 (Graph classification)**.** Given a graph G, graph classification is a function

$$f_{GC} : G \to C$$

that maps a graph one of the classes in set C.



```
Algorithm 1-WL (color refinement)
Input: $G = (V, E, X_V)$
1.  $c_v^0 \leftarrow \text{hash}(X_v)$ for all $v \in V$
2.  repeat
3.      $c_v^\ell \leftarrow \text{hash}(c_v^{\ell-1}, \{\!\{c_w^{\ell-1} : w \in \mathcal{N}_G(v)\}\!\})\ \forall v \in V$
4.  until $(c_v^\ell)_{v \in V} = (c_v^{\ell-1})_{v \in V}$
5.  return $\{\!\{c_v^\ell : v \in V\}\!\}$
```

Figure 2.1: Weisfeiler-Lehman isomorphism test (N. T. Huang and Villar, 2021)

Other common tasks in graph learning include community detection, graph generation, or graph matching, but the previously defined tasks are the most common for the area of this thesis.

2.3 Inductive and Transductive learning

In a transductive setting, a graph neural network can only make predictions on already seen data (A. Rossi et al., 2018). The test set is known, and nodes can be used for message passing (without using labels) during training. On the other hand, an inductive setting means the test set is not known during training, for example we mask some nodes of a graph, or predict on an unseen graph.

The choice of learning strategy depends on the specific dataset. Since transductive learning can access all features and complete graph structure, it can improve test set accuracy. Usually, however, inductive learning is more practical, since the test set is not always known and is more scalable, because it does not require retraining models each time a graph structure changes, unlike in the transductive setting.

2.4 Weisfeiler-Lehman (WL) graph isomorphism test

Weisfeiler-Lehman test is a powerful method to differentiate non-isomorphic graphs. It is analogous to message passing in graph neural networks (K. Xu et al., 2019).

Weisfeiler-Lehman isomorphism test starts with a single label for all nodes. Then, local neighborhoods are sampled and aggregated for each node, and a unique label is created using a hash function. The algorithm stops once there are no changes in labels between iterations. The complete algorithm is shown in Figure 2.1.

It was shown that message passing GNNs are at most equally expressive as the WL test. Therefore, topologically, message passing can only distinguish

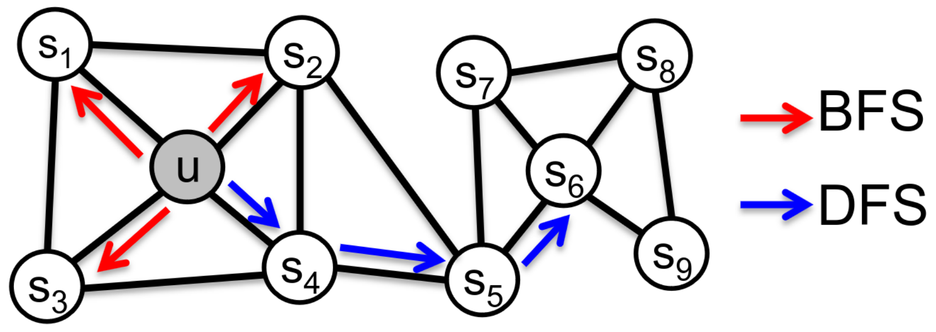


Figure 2.2: BFS and DFS search strategies from node u with walk length 3 (Grover and Leskovec, 2016).

the same structures as the WL test. This seems to be an issue only for datasets where node features are not crucial, and the model needs to learn mainly from the graph structure (K. Xu et al., 2019). Since most datasets rely on node feature information, this fact is relevant only for a subset of graph classification tasks, which need topological information.

2.5 Representation learning

High-quality features can have a significant impact on the performance of all machine learning models. For many real-world datasets, node (edge, graph) features can be extracted using application specific methods. For example, in citation networks, initial node features represent word embeddings of each paper's abstracts. Other areas relied on hand-crafted features specific for each individual dataset, such as node degrees, clustering coefficients, shortest-path lengths, and others. Universal algorithms for feature extraction can nevertheless provide a strong baseline.

Node2vec (Grover and Leskovec, 2016) is a popular unsupervised algorithm for learning node embeddings, which can be used as initial features for graph neural networks. Inspired by word2vec, it learns embeddings of nodes using the observation that nodes with similar neighborhoods should have similar embeddings.

Node2vec uses the concept of random graph walks. A random walk is a sequence of nodes where each next node is chosen randomly from the set of neighboring nodes. Node2vec then improves the idea by introducing two parameters, p and q, which control the probability of returning to the previous node or moving to a node further away. This allows the algorithm to explore both local and global neighborhoods. By changing the p, q parameters, node2vec effectively simulates local DFS or BFS search.

2.6 Graph Convolutional Networks

Based on their success in computer vision, (T. N. Kipf and Welling, 2017) have proposed a generalization of convolutional neural networks for the graph domain. Convolutions on images typically use a fixed grid mask, which cannot be applied to graphs, since the number of neighbors of each node can vary. The ordering of nodes in a graph is also arbitrary, which means that a permutation invariant function is needed.

Definition 2.5 (Permutation invariance)**.** Permutation invariant function is a function $f : \mathbb{R}^{|V|\times|V|} \times \mathbb{R}^{|V|\times m} \to \mathbb{R}^{|V|\times d}$, that satisfies

$$f(A, X) = f(PAP^T, PX)$$

for any permutation P (Leskovec, 2022).

Permutation invariant functions are required for graph-level tasks. In case of a node-level task, the function needs to be permutation equivariant, that is the output changes with the permutation of the input.

Definition 2.6 (Permutation equivariance)**.** Permutation equivariant function is a function $f : \mathbb{R}^{|V|\times|V|} \times \mathbb{R}^{|V|\times m} \to \mathbb{R}^{|V|\times d}$, that satisfies

$$Pf(A, X) = f(PAP^T, PX)$$

for any permutation P (Leskovec, 2022).

Graph Convolutional Networks (GCN) (T. N. Kipf and Welling, 2017) use the following propagation rule:

$$H^{(l+1)} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{2.1}$$

where:

- $\hat{A}$ is an adjacency matrix of the graph with added self-connections,
- $\hat{D}$ is a degree matrix of $\hat{A}$ and $(\hat{D}^{-\frac{1}{2}})^2 = \hat{D}^{-1}$,
- σ is a non-linear activation function
- H is a matrix of embeddings, W is a trainable weight matrix

A common example of a non-linear activation function is ReLU, defined by the following equation: $\text{ReLU}(x) = \max(0, x)$. Graph Convolutional Networks suffer from the fact that the trained model directly depends on a specific graph structure and, therefore, cannot be applied to a different graph. This limits the usage only to transductive tasks, which is very limiting. The propagation rule also relies on constructing full matrices, which is infeasible for most real-world datasets. However, the idea of convolutions on a graph was a strong inspiration for the message passing framework, which is the basis of most graph neural networks.

2.7 Message Passing Graph Neural Networks

Message passing is a general framework for information flow between nodes of a graph. It generally consists of three phases:

- Message computation
- Aggregation
- Update

Message computation is usually omitted as it corresponds to loading the node features. The general message passing equation for a graph neural network can be formulated as follows (Gilmer et al., 2017):

$$h_v^{(l+1)} = \gamma^{(l)} \left(h_v^{(l)}, \sum_{u \in \mathcal{N}(v)} f^{(l)} \left(h_u^{(l)}, h_v^{(l)}, e_{u,v}^{(l)} \right) \right) \tag{2.2}$$

where:

- $h_v^{(l)}$ represents node embedding of node v at layer l,
- $\mathcal{N}(v)$ denotes the neighborhood of node v,
- $e_{u,v}^{(l)}$ represents edge features between nodes u and v at layer l,
- f, γ are learnable functions (called *aggregation* and *update* respectively)

Different graph neural network layers can use a fixed function instead of learnable aggregation and update functions. Likewise, not all layers need to use edge features. The number of graph message passing layers is usually denoted as the depth of a graph neural network.

We can look at Graph Convolutional Networks in Section 2.6 through the lens of message passing. The propagation rule of GCN where d_i is a degree of node i can be rewritten as (T. Kipf, 2016):

$$h_v^{(l+1)} = \sigma \left(\sum_{j \in N(v)} \frac{1}{d_u d_v} h_v^{(l)} W^{(l)} + \frac{1}{d_v} h^{(l) W^{(l)}} \right)$$

In this case, aggregation is a weighted average of neighbor's embeddings and update function sums aggregated messages with the previous node representation.

Another example of a message passing neural network is GraphSAGE (Hamilton et al., 2018). An accomplishment of GraphSAGE is that it is possible to train it in an inductive setting, hugely extending the area of possible applications. GraphSAGE uses the idea of convolutions, but defines them over fixed-size local node neighborhoods, which allows more efficient training. It has the following message passing equation:

$$h_v^{(l)} = \sigma(W^{(l)} \cdot \mathrm{CONCAT}(h_v^{l-1}, AGG_{u \in N(v)}(h_u^{l-1}))) \tag{2.3}$$

where AGG is an aggregation function, such as mean, and CONCAT concatenates the vectors together.

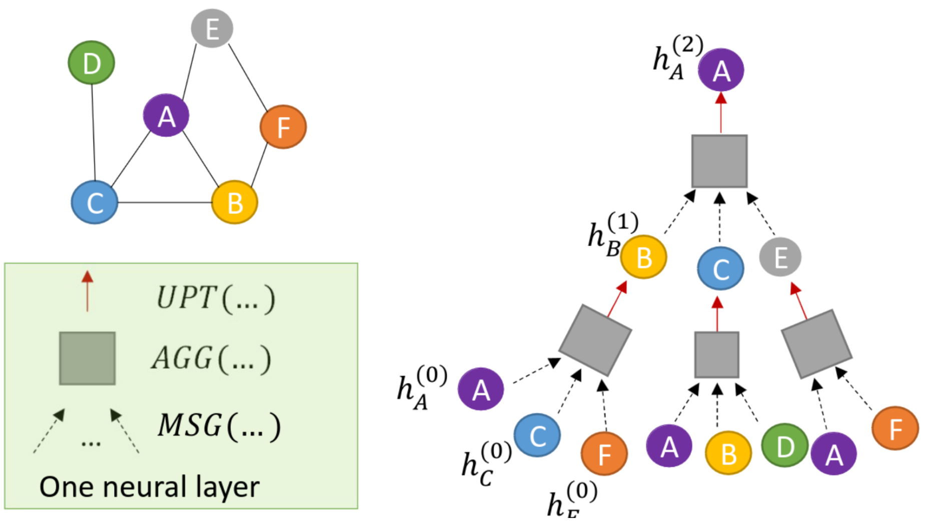


Figure 2.3: Computational graph of message passing. Input graph in the upper left corner and message passing from node A on the right (P. Li and Leskovec, 2022).

2.7.1 Attention layers

Attention mechanisms have achieved state-of-the-art performance in Natural Language Processing (NLP). (Veličković et al., 2018) extended the idea of self-attention to graphs and proposed the Graph Attention Network (GAT). The main idea of attention in graphs is to learn different weights for each node connection. This allows a more fine-grained control over the information flow in the network, since we can assign different importance to closer nodes, or nodes that are deemed more significant in the graph structure.

Computing the importance of node j features for node i is done using the attention mechanism below (Veličković et al., 2018):

$$e_{ij} = a\left(Wh_i, Wh_j\right) \tag{2.4}$$

a is a one-layer feed-forward neural network in the author's experiments, thus the complete attention mechanism is:

$$e_{ij} = \text{LeakyReLU}(a^T\left[Wh_i || Wh_j\right]) \tag{2.5}$$

And the message passing layer is:

$$h_i' = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij} \cdot Wh_j\right) \tag{2.6}$$

Where $\alpha_{ij} = \text{softmax}(e_{ij})$ is the normalized attention coefficient and LeakyReLU is the following function

$$LeakyReLU(x) = \max(0, x) + \text{negative\_slope} \cdot \min(0, x)$$

The original graph attention layer uses $negative\_slope = 0.2$. Operation $||$ in Equation 2.5 represents concatenation of vectors. Masked attention has a

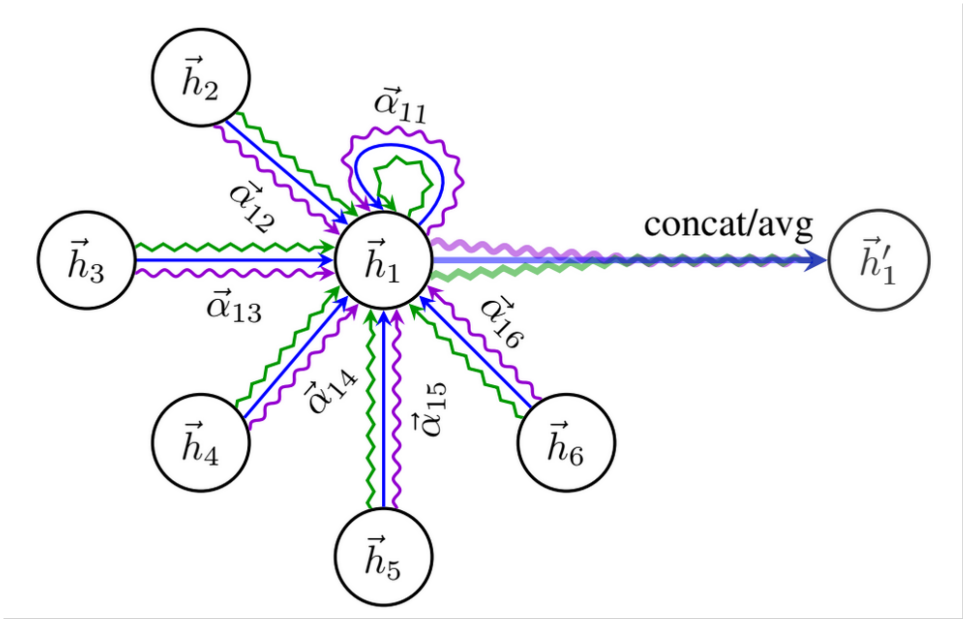


Figure 2.4: Graph Attention Network with multi-head attention (3 heads) (Veličković et al., 2018).

natural correspondence in graphs, as we compute the attention coefficients only for direct neighbors of a target node.

The authors have found that using multi-head attention is beneficial, similar to (Vaswani et al., 2023). Using multiple weight matrices (heads) and combining them leads to smoother learning (Veličković et al., 2018). Combination of multiple heads can be done by concatenation or averaging across all heads. Therefore, the final equation for the attention layer is:

$$h_i' = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i}\alpha_{ij}^{K}W^k h_j\right) \tag{2.7}$$

In the final layer, vectors don't need to be concatenated, but a different pooling function can be used, such as an average between all attention heads like Equation 2.7.

In (Brody et al., 2022), the authors have noticed a limitation of the original formulation of GAT (Veličković et al., 2018). The non-linear activation function (LeakyReLU in this case) is applied immediately after calculating the attention. They have shown that it leads to a collapse of linearities in W and a in Equation 2.5 to a single linear layer and proposed a fix by changing the order of operations to (Brody et al., 2022):

$$e_{ij} = a^T(\text{LeakyReLU}\left(W \cdot [h_i||h_j]\right)) \tag{2.8}$$

which separates the two layers by a nonlinearity and thus potentially increases the capabilities of the model.

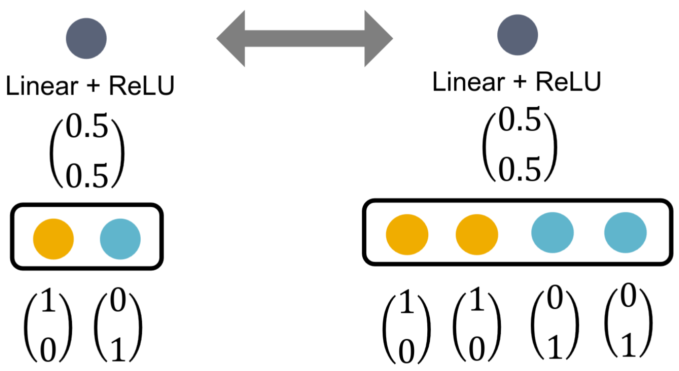

Linear + ReLU
$\begin{pmatrix}0.5\\0.5\end{pmatrix}$
$\begin{pmatrix}1\\0\end{pmatrix}$ $\begin{pmatrix}0\\1\end{pmatrix}$
Linear + ReLU
$\begin{pmatrix}0.5\\0.5\end{pmatrix}$
$\begin{pmatrix}1\\0\end{pmatrix}$ $\begin{pmatrix}1\\0\end{pmatrix}$ $\begin{pmatrix}0\\1\end{pmatrix}$ $\begin{pmatrix}0\\1\end{pmatrix}$

Figure 2.5: 2 graphs with one-hot encoded features will map to the same embedding vector when using GCN (Leskovec, 2022).

2.7.2 Graph Isomorphism Network

So far, in this chapter, only graph neural networks that mainly use features and edge weights to pass information between nodes were examined. Some tasks, however, require a deeper understanding of the graph structure itself. During message passing, computational graphs, which are shown in Figure 2.3, are being mapped to an embedding space (usually $\mathbb{R}^d$ for some positive integer d) using message passing layers (Leskovec, 2022). If this mapping is not injective, some computational graphs will be mapped to the same embedding vector. Therefore, ideally a graph neural network layer should be injective in this sense, as it could distinguish between different computational graphs. A layer is more expressive, if it can distinguish between more classes of graphs.

For example, given two graphs, with 2 and 4 nodes respectively, a Graph Convolutional Network will map both of the graphs in Figure 2.5 to the same embedding vector, despite the fact that they are different graphs, which hints at a low expressive power of the GCN. Graph Isomorphism Network (GIN) (K. Xu et al., 2019) is a graph neural network designed to be as expressive as the Weisfeiler-Lehman (WL) test, which was described in Figure 2.1.

The GIN layer is defined as follows:

$$h_v^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right)\cdot h_v^{(k-1)} + \sum_{u\in N(v)} h_u^{(k-1)}\right)$$

Where $\epsilon^{(k)}$ is a trainable parameter, or a fixed scalar, and $\mathrm{MLP}^{(k)}$ is a feed-forward neural network (K. Xu et al., 2019).

Using an expressive layer, such as GIN, is not required in many cases, as most datasets contain strong features, some of which can be topologically substantial. Graph Isomorphism Network is ideally suited for graphs with absent, or weak features where learning the topological structure is required to achieve sound performance.

2.8 Dataset sampling

Scalability of graph neural networks is one of the largest challenges of the field. In traditional machine learning, we can randomly split the dataset into mini-batches. In GNNs, due to message passing, embeddings of nodes depend on their neighbors, and the number of dependent nodes grows with each layer. This problem is called neighborhood explosion. Small graphs can be processed in a full-batch fashion, but most real-world datasets cannot be used this way.

If it is the case that the dataset is formed by many independent graphs, to increase training efficiency, multiple graphs can be stacked into a single graph. This works by diagonally stacking adjacency matrices. Since there are no edges across input graphs, message passing continues to function as intended, and no information is leaked across graphs (Fey and Lenssen, 2024). Adjacency matrix is generally implemented using sparse matrices, therefore no additional memory constraints are imposed by this operation.

Given adjacency matrices $\mathbf{A}_1, \dots, \mathbf{A}_n$ and feature matrices $\mathbf{X}_1, \dots, \mathbf{X}_n$, the stacked graph $\mathbf{A}$ is:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \qquad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix},$$

To combat the issue of neighborhood explosion, in GraphSAGE (Hamilton et al., 2018), the authors have proposed the idea of sampling the neighborhood, instead of using all neighbors. At each layer, only a fixed amount of nodes is expanded, which is shown in Figure 2.6. Neighborhood sampling is a trade-off between memory efficiency and information loss. Since not all neighbors of a node are gathered for message passing, some information is necessarily lost in the process.

Cluster-GCN (Chiang et al., 2019) advances the idea of neighborhood sampling introduced by GraphSAGE. They noticed the importance of using limited neighborhoods, but instead of random sampling, they propose first clustering the graph, and then use clusters as mini-batches. This yields high reuse of calculation within a mini-batch, because neighbors within the batch are common and eliminates the risk of neighborhood expansion problem. Despite improving the sampling efficiency, it introduces the need for a clustering algorithm, which can slow down the training pipeline and possibly introduce additional hyperparameters.

2.8.1 Oversmoothing

Oversmoothing is an issue that occurs during the training of graph neural networks. If a trained graph neural network has many layers (the exact number depends on the specific dataset), message passing will cause each node to get information from far away nodes. Thus, since messages are

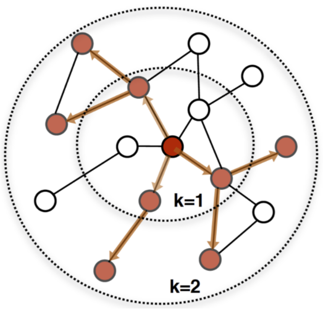


Figure 2.6: Sampling neighborhoods in GraphSAGE (Hamilton et al., 2018), k is the number of layers.

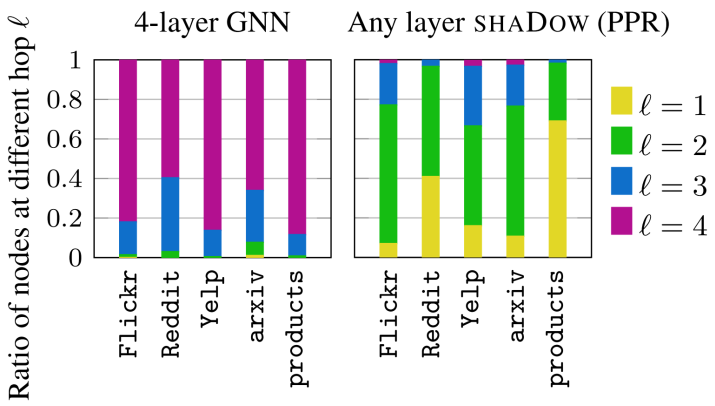


Figure 2.7: Neighborhood composition of GNN vs ShaDow across different datasets (Zeng et al., 2021).

propagated for all nodes, node features can converge to a similar area in the embedding space, despite being fundamentally different.

ShaDow-GNN (Zeng et al., 2021) approaches the issue of oversmoothing by extracting a neighborhood subgraph G_v and using a graph neural network directly on this subgraph. It allows a design of a GNN with more layers than the depth of G_v (distance of the farthest node from the center node) without gathering messages from nodes far away. This leads to a local oversmoothing on G_v, which can be beneficial, as it corresponds to the extraction of higher-level features.

Figure 2.7 shows the neighborhood composition of a traditional GNN and ShaDow-GNN. Neighborhood composition is the number of nodes sampled at various distances from the current node. Because of neighborhood explosion, most of the nodes sampled during regular GNN training are four hops away, as that is the number of GNN layers used. ShaDow on the other hand, samples mainly close neighbors, and thus supports strong local message exchange, which is beneficial if it is assumed that the graph structure contains useful information.

Chapter 3

Temporal Graph Neural Networks Review

3.1 Introduction

The majority of graph machine learning algorithms focus on static graphs. Real data, however, is in many cases dynamic. Examples include social networks, where the network graph is continuously updated with new interactions by users, computer networks, which collect information in real-time and others. In areas such as citation graphs, value lies in seeing the research trends, and their evolution throughout time. Historical information can also be useful in predicting future relations, therefore the study of temporal graph neural networks is of great importance.

Fundamentally, there are two ways to define dynamic graphs. Discrete-time dynamic graphs (DTDG) are sequences of static graphs, while the more general Continuous-time dynamic graphs (CTDG) represent a graph as a sequence of addition and deletion events of nodes and edges.

Depending on the specific subfield, certain authors may prefer to use the term *dynamic* over *temporal*. For clarity, the terms *dynamic graph* and *temporal graph neural network* are preferred in this work. They can be, however, used interchangeably and can be used as such in this thesis.

3.1.1 Definitions

Definition 3.1 (Discrete-time dynamic graphs (DTDG))**.** A discrete-time dynamic graph is a sequence of static graphs Theorem 2.1, called snapshots in this context, $G = [G_1, G_2, G_3, \ldots G_n]$, where n is the number of distinct timestamps.

Definition 3.2 (Continuous-time dynamic graph (CTDG))**.** A sequence of time-stamped events (interactions) $G = \{x(t_1), x(t_2), \ldots x(\tau)\}$, where $x(t_i)$ represent an addition, deletion of a node (node event) or a directed edge (edge event), or a feature change at time t_i (E. Rossi et al., 2020).

3.1.2 Temporal learning tasks

Definition 3.3 (Temporal node classification)**.** Given a dynamic graph G with a set of nodes V, the temporal node classification task consists of learning a function f_{TNC}:

$$f_{TNC} : V \times \mathbb{R}^{+} \rightarrow C$$

which maps a node to a set of possible node classes C at time $t \in \mathbb{R}^{+}$ (Longa et al., 2023).

Definition 3.4 (Temporal link prediction)**.** Given a dynamic graph G with a set of nodes V, the temporal link prediction task consists of learning a function f_{TLP}:

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

which predicts a probability of an edge existing between two nodes at time $t \in \mathbb{R}^{+}$ (Longa et al., 2023).

3.2 Discrete-time dynamic graphs

Generally, discrete-time dynamic graph problems can be solved using a combination of a classic graph neural network to model relations within each static graph and a sequence model to model features evolving over time.

More formally, sequence models, given an input a sequence of observations $\{x^{(1)}, x^{(2)}, \dots, x^{(\tau)}\}$ where $x^{(t)} \in \mathbb{R}^{d}$ for all $t \in \{1, \dots, \tau\}$, produce output hidden representations $\{h^{(1)}, h^{(2)}, \dots, h^{(\tau)}\}$ where $h^{(t)} \in \mathbb{R}^{d'}$ for all timestamps $t \in \{1, \dots, \tau\}$. Here, τ represents the length of the sequence or the timestamp for the last element in the sequence (Kazemi, 2022).

The hidden representation $h^{(t)}$ is computed by the following equation:

$$h^{(t)} = \mathrm{RNN}(x^{(t)}, h^{(t-1)}) \tag{3.1}$$

where RNN is a specific sequence model.

One of the simplest ways to model dynamic graphs is to combine a recurrent neural network and a graph neural network.

Given a DTDG with a sequence of graph snapshots $[G^{(1)}, \dots , G^{(\tau)}]$, a GNN is applied to each graph snapshot, resulting in a sequence of embeddings $[Z^{(1)}, Z^{(2)}, \dots , Z^{(\tau)}]$, which is then applied as an input to an RNN Equation 3.1, that produces the final embeddings (Kazemi, 2022).

In ROLAND (You et al., 2022), the authors proposed an altered update method, which uses not only embeddings produced by the final layer of a GNN, but all the representations gathered in the intermediate layers of a GNN. The produced embeddings thus directly depend on the last layer as well as on the embedding in the last timestamp. The algorithm is shown in Algorithm 1.

3.3 Continuous-time dynamic graphs

Definition 3.5 (Temporal neighborhood)**.** Temporal neighborhood of node i at time interval T is a set of nodes $N_i(T) = \{j : (i, j) \in E(T)\}$ where $E(T)$ are (directed) temporal edges that exist at time interval T (E. Rossi et al., 2020).

Algorithm 1 ROLAND(You et al., 2022)

```
Input: Dynamic graph snapshot $G_t$, hierarchical node state $H_{t-1}$
Output: Node state $H_t$
1: $H_t^{(0)} \leftarrow X_t$
2: for $l = 1, \dots, L$ do
3:     $\widetilde{H}_t^{(l)} = \mathrm{GNN}^{(l)}(G_t, H_t^{(l-1)})$
4:     $H_t^{(l)} = \mathrm{Update}^{(l)}(H_t^{(l-1)}, \widetilde{H}_t^{(l)})$
5: end for
```

3.3.1 Temporal Message Passing

The message passing schema can be extended to dynamic graphs by using the notion of the temporal neighborhood. First, messages are created by nodes that were part of an interaction.

Because Theorem 3.2 uses directed edges, there will be two messages in case of an edge event. Messages are computed using a learnable message function *msg* for the source node and target node, respectively (E. Rossi et al., 2020):

$$\hat{h}_i^{(t)} = \mathrm{msg}(h_i^{(t-1)}, h_j^{(t-1)}, \Delta t, e_{ij}(t)) \tag{3.2}$$

$$\hat{h}_j^{(t)} = \mathrm{msg}(h_j^{(t-1)}, h_i^{(t-1)}, \Delta t, e_{ij}(t)) \tag{3.3}$$

In case of a node event $v_i(t)$, only a single message is computed for the involved nodes (E. Rossi et al., 2020):

$$\hat{h}_i^{(t)} = \mathrm{msg}(h_i^{(t-1)}, t, v_i^{(t)}) \tag{3.4}$$

The previous equations generate only the so-called memory embedding of a node. To get the actual temporal embedding $z_i(t)$ of node i at time t, an embedding module is used, which aggregates messages directly from all its temporal neighbors (E. Rossi et al., 2020):

$$z_i^{(t)} = \sum_{j \in N_i^k(t)} \phi(h_i^{(t)}, h_j^{(t)}, e_{ij}, v_i^{(t)}, v_j^{(t)}) \tag{3.5}$$

where ϕ is a learnable function, $N_i^k(t)$ is the k-hop temporal neighborhood of node i at time t, $v_i^{(t)}$ is the feature of node i at time t and e_{ij} is the edge feature between nodes i and j. $z_i^{(t)}$ is the final temporal embedding, which can be combined with a learnable time encoding to signify importance of the most recent interactions.

Equation 3.5 can be seen as a generalization of the message passing equation in Equation 2.2 to dynamic graphs.

In Temporal Graph Networks (TGN) (E. Rossi et al., 2020), the authors use a recurrent neural network, specifically a GRU, to calculate current memory state of each node, by combining the previous state and the received message. They experiment with different embedding modules to produce the final

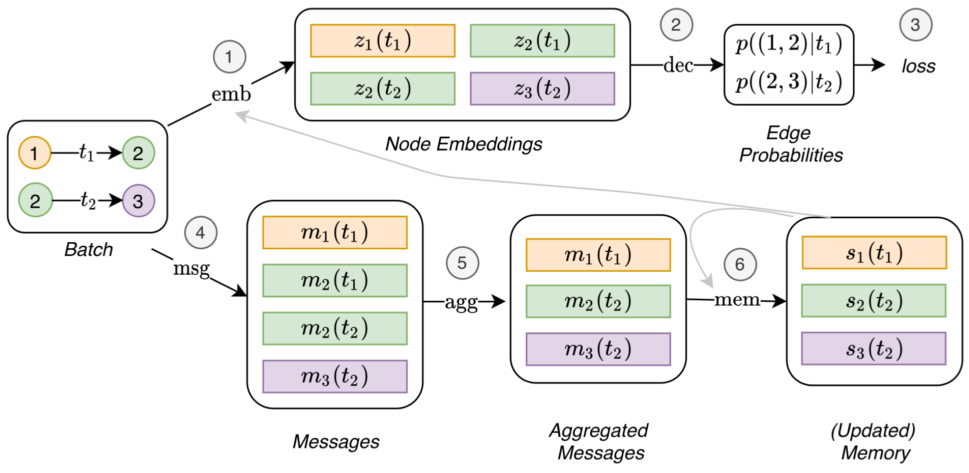


Figure 3.1: Illustration of the Temporal Graph Network architecture with an optional decoder for edge predictions (E. Rossi et al., 2020).

temporal embedding, and discover that using graph attention layers achieves best results (E. Rossi et al., 2020).

To achieve faster training, in TGN the authors use batches of events (interactions). Multiple events involving a single node or an edge can occur in a single batch. An aggregation (agg) module is thus used to combine multiple messages into one. The process is shown in Figure 3.1. In experiments, they found that averaging messages of events performs better than using only the last event, but incurs a higher computation cost.

3.3.2 Temporal Graph Attention layer

In TGAT (D. Xu et al., 2020), the authors extend the graph attention layer from static to dynamic graphs by combining temporal neighborhood information with time embedding for the source node.

For the time embedding, the important factor is relative time distance, not absolute time. It can thus be calculated using a difference between a source node and its neighbors, for example, given node embeddings of source node h_0 at time t and layer $(l-1)$, and neighbor embeddings

$$\{\tilde{h}_1^{(l-1)}(t_1), \dots, \tilde{h}_N^{(l-1)}(t_N)\}$$

the time embedding will be calculated using a (learnable) function with inputs

$$\{t - t_1, \dots, t - t_N\}$$

that represent the time difference between the source node timestamp and the last interaction timestamp of its neighbors. The time embedding is then combined with node embeddings to produce an input to the attention layer. To incorporate edge features, in TGAT the authors concatenate edge features with node embeddings and time embeddings.

Chapter 4
Implementation Details

To implement code for the experiments, Python was used, along with popular machine learning libraries.

Pytorch and Pytorch Lightning

Pytorch (Ansel et al., 2024) was used as the main deep learning backend, which provides basic building blocks for neural networks, such as activation functions, linear layers and optimizers.

To streamline the training process, for static models (Pytorch) Lightning (Falcon and The PyTorch Lightning team, 2019) was used. It is a layer on top of Pytorch, which emphasizes modular architecture, while providing additional features such as logging, checkpointing models, and callbacks such as learning rate schedulers and early stopping. It makes it manageable to write code that runs on multiple hardware backends without any changes.

All models are wrapped in a `LightningModule` class, equivalently the datasets are wrapped in a `LightningDataModule` class. This allows easy integration with Lightning's `Trainer` class, which handles the training loop. It is important to note that at any point, the layers of abstraction of the Lightning library can be overridden and Pytorch can be used directly.

Furthermore, accompanying `LightningCLI` was used to run experiments. When used with other Lightning modules, it allows for straight-forward configuration of arguments from a command line, or through a `YAML` configuration file. An example of such configuration is shown in Listing 4.1.

Listing 4.1: Example of a Lightning YAML configuration file

```yaml
data:
  root: /Datasets/OGB/
  batch_size: 8192
model:
  class_path: src.models.gcn.GCN
  init_args:
    hidden_channels: 256
    num_layers: 3
    dropout: 0.5
    norm: graph
```

Pytorch Geometric

To implement graph neural networks, Pytorch Geometric (PyG) (Fey and Lenssen, 2019), was used. PyG provides abstractions on top of Pytorch, which can be used to create specific graph layers and utilities. It comes with a variety of implemented graph convolutional layers, data loaders, graph transforms and aggregation operators among others. PyG tries to use sparse tensors as much as possible to reduce memory footprint in comparison to using dense tensors, thus making it possible to process larger graphs. Some quality-of-life improvements for graphs include the ability to define complicated model structures with residual connections, or different layer inputs using PyG's `Sequential` class, showcased in Figure 4.1.

```
model = Sequential(
    "x, edge_index, batch",
    [
        (GCNConv(num_features, 64), "x, edge_index -> x1"),
        ReLU(),
        (GCNConv(64, 64), "x1, edge_index -> x2"),
        ReLU(),
        (lambda x1, x2: [x1, x2], "x1, x2 -> xs"),
        (JumpingKnowledge("cat", 64, 2), "xs -> x"),
        Linear(128, num_classes),
    ]
)
```

Figure 4.1: Example of PyG's `Sequential` class

GraphGym

GraphGym (You et al., 2021) is a library designed to evaluate and compare different configurations of GNN modules. It provides a modular design, which makes it easy to run many experiments and perform grid search of selected combinations of parameters. Although it is coupled with Pytorch Geometric and promises to be easily extendable, I have found it cumbersome when applying to custom datasets and using non-standard evaluation settings. For example, splitting the datasets into multiple folds for cross-validation was not possible by default. This led me to write my own training pipeline. However, I recognize the usefulness of GraphGym when designing new layers, as those can be well integrated, tested, and compared with other configurations on the plethora of existing datasets.

NetworkX

NetworkX[1] is a Python library used for network analysis and manipulation of graphs of any kind. In this work, it has been used to load and preprocess input graphs and to output graphs for further visualization. Written in Python, its usage is limited to small graphs, due to the performance limitations. Conversely, feature completeness provides a compelling argument for its usage in light computational tasks. One of the issues I have found working with NetworkX is that the conversion of a NetworkX graph to a Pytorch Geometric graph requires all nodes and edges to have the exact same features. This was a source of many hiccups during implementation. A workaround was to delete incomplete features from the graph. Another way to solve this issue would be to use a heterogeneous PyG graph.

Available benchmark datasets

Absence of high-quality datasets has been an issue in developing graph neural networks. Many papers have used datasets such as Cora, CiteSeer or PubMed (Yang et al., 2016) for benchmarking results. These datasets only contain a few thousand nodes and tens of thousands of edges. However, many real-world datasets contain millions of nodes and edges. This issue has sparked interest and led to the development of Open Graph Benchmark (OGB) (Hu, Fey, Zitnik, et al., 2021). OGB is a collection of datasets varying in size, task type, and domain. To further improve the development of GNNs, authors of OGB have since provided even larger datasets, the Large Scale Graphs (OGB-LSC) (Hu, Fey, Ren, et al., 2021), which contain up to 240M nodes and 1.7B edges and show that GNNs can be successfully used at this scale (Hu, Fey, Ren, et al., 2021). After recognizing the same need for dynamic graphs, similar efforts are being made in the Temporal Graph Benchmark (TGB) (S. Huang et al., 2023).

Libraries for temporal graph neural networks

Although multiple well-established libraries exist for static graph neural networks, the same cannot be said for temporal graph neural networks. How to represent dynamic graphs to allow efficient sampling and training is still an open question. Most temporal graph neural networks are implemented in pure Pytorch, without using the latest advances in static GNN libraries. As an example, to perform the experiments on dynamic graphs, I have used the DyGLib-TGB (Yu, 2023), because it is the recommended repository by the authors of TGB. Although an extension of PyG to temporal graph neural networks, called Pytorch Geometric Temporal[2] exists, the development has been discontinued. Efforts are underway to add proper support for temporal datasets and models in Pytorch Geometric, but as of version 2.5, it is still in early stages. The lack of efficient tooling for temporal graphs is, in my

[1]https://networkx.org/

[2]https://pytorch-geometric-temporal.readthedocs.io/en/latest/

opinion a significant obstacle in the development of temporal GNNs and remains to be solved by future work.

Experiment tracking

Weights and Biases[3] was used for logging and experiment tracking. It is well integrated with Lightning and can be used to track hyperparameters, metrics during training and even store artifacts of models and datasets. Many visualizations are readily available, with options to export data to perform local analysis.

[3]https://wandb.ai

Chapter 5

Proposed Method and Assesment

This chapter will discuss the proposed solution for the tasks of node classification on static and dynamic graphs. Section 5.1 will describe the proposed algorithms for static node classification, while Section 5.2 will discuss the designs for temporal node classification.

5.1 Static graphs

5.1.1 Dataset

To evaluate performance of classical graph neural networks, a dataset of multiple static graphs was chosen. Each graph in this dataset represents a wall inscribed with an ancient hieroglyphic text.

Figure 5.1: Picture of one of the hieroglyphic walls in Egypt.

The hieroglyph text is transformed into a facsimile, a digital re-drawing of the glyphs. The facsimiles were then processed by a segmentation algorithm and several correction steps to create a graph representation. The exact algorithms and methods are described in detail in (Mařík et al., 2024).

The nodes represent individual hieroglyphs, as either centers of glyphs, or touch points, the closest points between two glyphs. The task is to classify if a node in the graph is a part of a hieroglyph, or not. Accurate predictions would help to reconstruct the reading sequence of hieroglyphs, which is crucial for end-to-end translation. Part of an input graph is visualized in Figure 5.2b. The learning task is an inductive node classification, because the model is evaluated on unseen graphs.

Dataset statistics

The dataset consists of 7 graphs, with an average of 18995 nodes and 24186.14 edges directed edges. The label ratio is skewed towards negative labels, with an average number of 31.8% positive labels (hieroglyph nodes) in each graph. The exact statistics are shown in Table 5.1.

Table 5.1: Hieroglyphs graph dataset statistics.

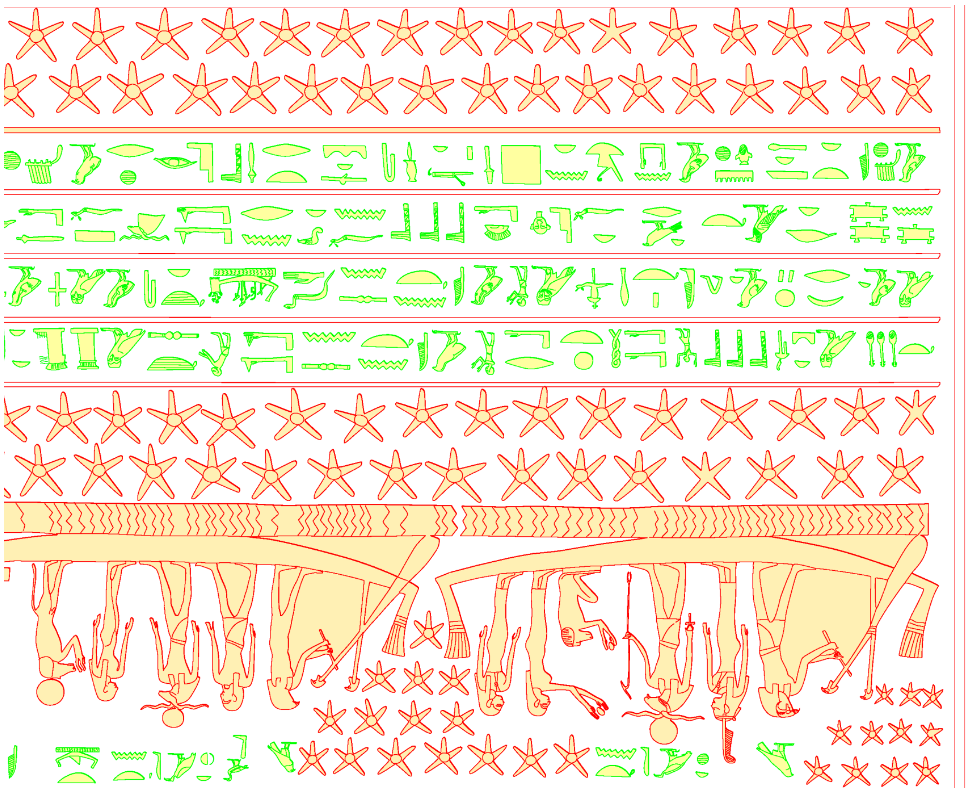
Graph	# Nodes	# Edges	Percentage of Positive Labels
1	19836	25419	53%
2	26891	34392	27%
3	29608	37879	28%
4	6191	7929	39%
5	9429	12003	30%
6	21987	27929	24%
7	19023	23752	22%

Preprocessing

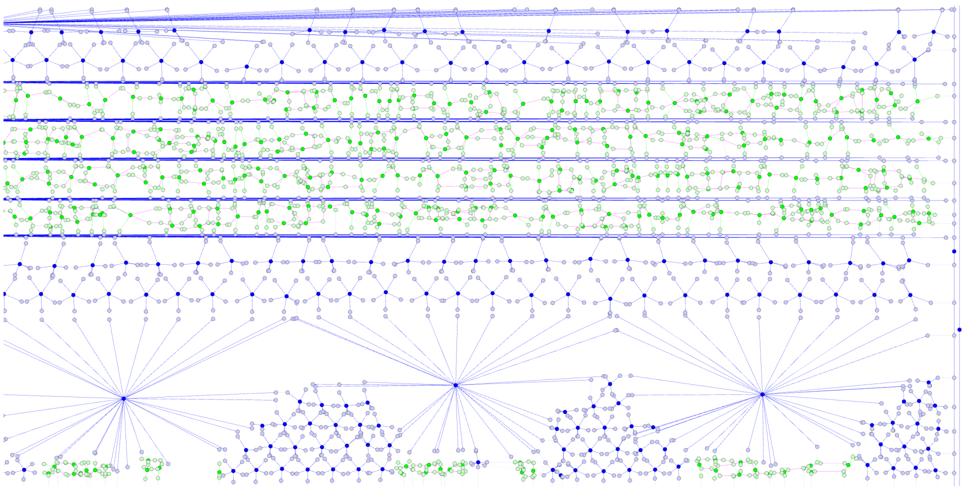
Dataset is ingested from `json` files, which contain information about center points of glyphs, touch points and edge data, along with a `gml` file which represents the graph structure. The input dataset contains labels, which specify a node class, which can be one of parts of hieroglyph symbol, vertical or horizontal stripe (delimiter) or other insignificant glyph. Since the goal is to classify if a node is a part of a hieroglyph, labels were converted to be either 0 or 1, where 1 represents a part of a hieroglyph, therefore making the classification task binary. Features and graph structure were then combined to create a complete graph dataset.

Edge features

Aside from node features, many graph neural networks can directly use (multidimensional) edge features during message passing. The dataset includes edge features, which could be used during training, however not all edge



(a) : Image of a labeled facsimile of hieroglyphic text, with green glyphs representing hieroglyphs.



(b) : Graph representation of Figure 5.2a. Green nodes are parts of hieroglyphs. Visualization created using Gephi[a].

[a]https://gephi.org/

Figure 5.2: Inputs to the model, hieroglyphic text page and its graph representation.

features are present for all edges. This presents an implementation problem, since PyTorch Geometric requires all edges and nodes to have the exact same number of features. Filling in such missing features with zeros was considered, but was found to slow down training, without providing any benefits. Other approach to solve this issue would be rethinking the underlying graph data representation and using a heterogeneous graph, along with a heterogeneous graph neural network, but such experiments were outside the scope of this thesis.

5.1.2 Baseline models

To compare the effects of providing models with graph structure information, a baseline Multilayer Perceptron (MLP) was used. Multilayer perceptron is a neural network consisting of fully connected (linear) layers, with a non-linear activation function, such as ReLU, applied between linear layers.

As a second baseline model, a Graph Convolutional Network (GCN), defined in Section 2.6, was used. Comparisons with the GCN should provide insights into whether more complicated models are necessary, or if using the graph structure yields the most significant improvement. The task is inductive in terms of seeing new graphs during testing, therefore GCN can be used.

5.1.3 Proposed models

Because the dataset has different types of nodes (centers, touch points and separators), edges between nodes should not be equally important. This proposition would be confirmed by performance difference between a baseline GCN, which uses a constant coefficient for all edges, and a more complicated model, such as GAT, which can learn weights for edges.

Graph Attention Networks, described in Subsection 2.7.1, have been shown to outperform GCNs on many tasks, therefore models using attention layers were tested. Because (Brody et al., 2022) have shown that the original attention layer design proposed in (Veličković et al., 2018) might be suboptimal, both the original GATConv and newer GATv2Conv layers were used in experiments. The GATv2Conv promises stronger expressiveness by including a non-linear activation function in the attention mechanism.

To evaluate if using topological information is necessary for strong performance on this dataset, a Graph Isomorphism Network (GIN) (K. Xu et al., 2019), described in Subsection 2.7.2, was used. If GIN outperformed models using attention layers, or graph convolutional layers, it would suggest that node and edge features are insufficient to represent the mentioned graph. Furthermore, a combination of attention layers and GIN was used to test if the inclusion of both mechanisms could provide more robust performance, by ideally combining the strengths of both methods.

In general, the order of layers in a graph neural network follows the schema in Figure 5.3a. A chosen graph convolutional layer is followed by normalization, activation and dropout, which is repeated for the number of layers before being passed to a final fully-connected linear layer to produce outputs. All evaluated GNN models, except DeeperGCN, in this experiment follow this schema.

To evaluate one of the recent advancements in graph neural network architecture, a DeeperGCN (G. Li et al., 2020) model was tested. DeeperGCN uses residual connections between layers and in (G. Li et al., 2020) the authors find an ordering of modules that improves performance. They propose an ordering, which is shown in Figure 5.3b. Using residual connections and a new convolutional layer they managed to successfully scale the model to hundreds of layers without encountering oversmoothing, discussed in Subsection 2.8.1.

Normalization

To improve model convergence, normalization layers were used, specifically a GraphNorm layer (Cai et al., 2021), which normalizes feature j of node x_i using the formula:

$$\text{GraphNorm}(x_{i,j}) = \gamma_j \cdot \frac{x_{i,j} - \alpha_j \cdot \mu_j}{\sigma_j} + \beta_j \tag{5.1}$$

where μ_j is the mean of the j-th feature, and

$$\sigma_j^2 = \frac{\sum_{i=1}^{n} \left(\hat{h}_{i,j} - \alpha_j \cdot \mu_j\right)^2}{n}$$

α, β, and γ are learnable parameters, and n is the number of nodes in the graph.

Training

Cross entropy loss, defined in Equation 5.2 was used to optimize the model, optimizer of choice was the standard Adam optimizer (Kingma and Ba, 2017), with an option to reduce learning rate when the training loss stopped decreasing. Dropout layers, which set neuron weights to zero with a given probability, were used to prevent overfitting.

$$\text{Cross Entropy}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{n} y_i \log \hat{y}_i \tag{5.2}$$

where $\mathbf{y}$ is the true label, and $\hat{\mathbf{y}}$ is the predicted label.

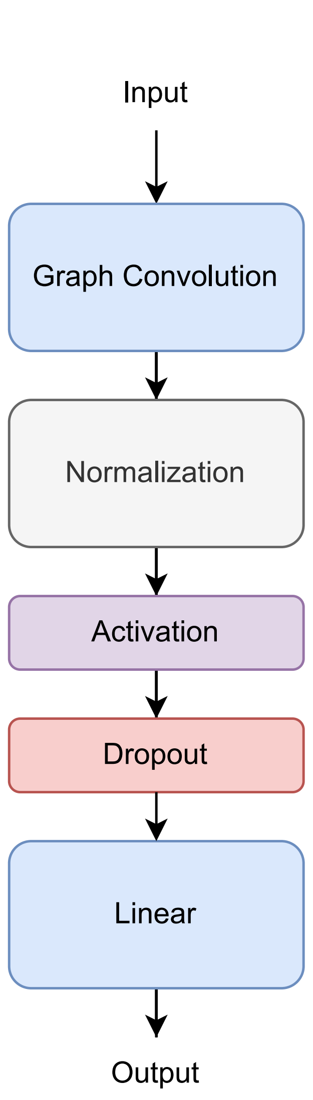
Metrics

Standard metrics were used to evaluate the performance, namely accuracy:

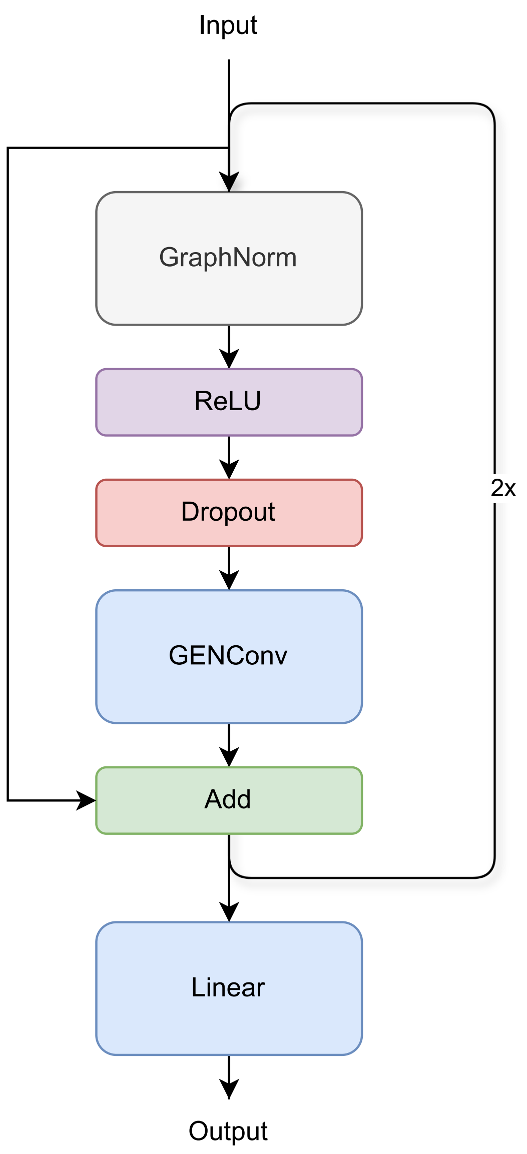
$$\text{Accuracy} = \frac{\#\ \text{Correct classifications}}{\#\ \text{All classifications}} \tag{5.3}$$

and the area under the receiver operating characteristic curve (AUROC). ROC shows the true positive rate

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$



(a) : A diagram of a general composition of layers in a graph neural network.



(b) : A diagram of the DeeperGCN model used for the static experiments.

Figure 5.3: Diagrams of graph neural network layer compositions.

against the false positive rate

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}}$$

and the area under the ROC samples different threshold to produce a scalar for comparisons.

5.2 Dynamic graphs

5.2.1 Dataset

Performance evaluation of temporal graph neural networks was done in two parts. First, to verify reported performance metrics, selected models were tested on a standard benchmark dataset, the Temporal Graph Benchmark (TGB) (S. Huang et al., 2023) was used. Specifically a dynamic node classification dataset `tgbn-trade`, which is a dataset of a trading network between nations, with edges representing the yearly trade value from one nation to another. The task is to predict the future trade ratio value between all nations(S. Huang et al., 2023).

Selected temporal models were then evaluated on a dataset of a citation network of scientific papers, `ogbn-arxiv`, which is a benchmark dataset for static graph neural networks. The dataset was transformed into a Continuous-time dynamic graph Theorem 3.2. Conversion of the original dataset to a Discrete-time dynamic graph Theorem 3.1 and usage of corresponding models was considered, but my main interest in temporal methods is in continuous-time graphs, as they can be more generally used.

`Ogbn-arxiv` is a dataset in the Open Graph Benchmark (OGB) (Hu, Fey, Zitnik, et al., 2021). It is a graph representing papers from arXiv, in the area of Computer Science. Each node has a feature vector, which corresponds to a word embedding of each paper's abstract. A directed edge is present if one paper cites another. The task is to predict one of 40 subject areas to which a paper belongs. It is important to note that the labels are noisy, as a paper can lie at an intersection of multiple areas.

The idea behind this experiment was to evaluate whether temporal GNNs can match the performance of static GNNs. Furthermore, correct predictions of node classes could help with analysis of research trends, which is poised to become a crucial area of study given the increased amount of research papers. Temporal models are able to produce node embeddings at every timestamp, which could be used for dynamic clustering to track evolution of the citation network.

Dataset statistics

Summary statistics of the used datasets are shown in the tables below, `tgbn-trade` in Table 5.2 and `ogbn-trade` in Table 5.3.

Table 5.2: Dataset statistics of `tgbn-trade`.

# Nodes	# Edges	# Timestamps
255	468245	31

Table 5.3: Dataset statistics of `ogbn-arxiv` after conversion to a temporal dataset.

Data	# Nodes	# Edges	# Timestamps
Train	87599	374839	25
Validation	30238	622466	1
Test	51506	1166243	2

Preprocessing

To convert the original `ogbn-arxiv` dataset into a continuous-time dynamic graph, I have extracted timestamps of edges by looking at edge creation as the time of paper citation. Therefore, to prevent leakage of information from the test set, timestamp of an edge is the year in which the later paper was published. This ensures that an edge is created only at a time of the later paper. The learning task is therefore considered to be inductive. Interaction features (messages in terminology of PyTorch Geometric) were created by concatenating both node creation times.

The dataset was then split into training, validation and test sets by time, the same way as the original static dataset. During temporal node classification, temporal models use all edges from previous timestamps (including those in training) to perform message passing and aggregation, but are evaluated only on nodes in the test or validation set, which is a standard practice.

5.2.2 Baseline models

Since the main idea behind the experiment is to evaluate if using temporal information with temporal graph neural networks is beneficial, baseline models are classical graph neural networks, without access to time data. Similarly to experiments on static graphs in Section 5.1, the two baseline models are a Multilayer Perceptron (MLP) and a Graph Convolutional Network (GCN). These models are therefore evaluated on the original `ogbn-arxiv` static dataset.

As a baseline for the `tgbn-trade` dataset, a Moving Average is used. Moving Average simply averages node labels seen at previous timestamps. It is therefore usable only in transductive setting, since it relies on observing previous node labels. This baseline has proven to be a strong contender on TGB datasets, outperforming all temporal graph neural networks.

5.2.3 Selected temporal models

The subsequent temporal models were selected for evaluation:

- Temporal Graph Network (TGN) (E. Rossi et al., 2020), which represents a memory-based model to store node embeddings.
- Temporal Graph Attention (TGAT) (da Xu et al., 2020), which uses an attention layer with temporal message passing.
- DyGFormer (Yu et al., 2023), a model leading in many TGB benchmarks, which combines attention with neighbor co-occurrence to encode common historical neighbors.

DyGFormer (Yu et al., 2023) is a state-of-the-art temporal graph neural network, the best performing model on several TGB datasets. One of the improvements the authors claim is that only one-hop historical neighborhood is extracted, which is then combined with edge and time embeddings. They also consider the frequency at which nodes appear in the same one-hop neighborhood, which they call neighbor co-occurrence. This feature is poised to be helpful in link prediction tasks, because it signifies which nodes are likely to be close in the graph. The neighbor co-occurrence features are then fed into the attention layers along with node, edge and time embeddings.

Batch size

It is important to note that for a fair comparison, all temporal graph neural networks should be used with the same batch size, because changing the batch size can lead to different results, which was argued in TGN (E. Rossi et al., 2020). When using a batch size larger than one, there is a chance that a single node will appear in multiple interactions. Many strategies for aggregation into a single message exist, such as averaging, using only last interaction, or a more complicated aggregation function such as using a recurrent neural network. For temporal models this is a trade-off, as higher batch size improves training speed, but can lead to forgetting because the aggregation function is unable to reduce multiple interactions into one message. The detailed diagram showing parts of this process was shown in Figure 3.1.

Metrics

To reproduce the results of the original papers, the same metrics are used on the `tgbn-trade` dataset. It uses the Normalized Discounted Cumulative Gain (NDCG) score, which is a measure of ranking quality, often used in recommendation settings.

Discounted Cumulative Gain (DCG) (Järvelin and Kekäläinen, 2002) calculates the relevance of predicted scores $G = (x_1, x_2, \dots, x_n)$ by penalizing later predictions. The formula for DCG is:

$$\mathrm{DCG} = \sum_{i=1}^{n} \frac{x_i}{\log_2(i+1)}$$

Usually, only k most relevant predictions are accounted for, denoted as DCG@k (Järvelin and Kekäläinen, 2002).

$$\text{DCG@k} = \sum_{i=1}^{k} \frac{x_i}{\log_2(i+1)}$$

Normalized DCG (NDCG) is then obtained by dividing DCG by the IDCG(Järvelin and Kekäläinen, 2002),

$$\text{IDCG} = \sum_{i=1}^{n} \frac{2^{x_i} - 1}{\log_2(i+1)}$$

which assumes perfect ranking scores.

$$\text{NDCG@k} = \frac{\text{DCG@k}}{\text{IDCG@k}} \tag{5.4}$$

All TGB datasets then use the NDCG@10 metric.

The evaluation on the `ogbn-arxiv` dataset uses standard accuracy, defined in Equation 5.3.

Chapter 6
Experiments

This chapter discusses the experimental results of the proposed models. Section 6.1 presents the results of experiments on the static hieroglyph dataset. Section 6.2 deals with temporal datasets, assesses the performance of temporal graph neural networks on a previously used dataset and applies them to a novel dynamic node classification task.

6.1 Static graphs

The performed experiment on the hieroglyph dataset was a binary classification task. Given an input graph, the task was to predict whether the nodes are a part of a hieroglyph or not. Because the model is trained and evaluated on distinct graphs, this task is inductive.

6.1.1 Hyperparameters

To facilitate reproducibility of experiments, used hyperparameters are listed here. The batch size was 1, to train at one graph at a time. The Adam optimizer was used, learning rate was set to 0.01, with a reducing factor 0.1 whenever loss function stopped decreasing for at least 5 epochs. Number of hidden channels to 32, all layers used 3 convolutional layers (except for the MLP). Dropout was set to 0.5, edge feature dimension was 1. Attention layers used 4 heads, and results of each head were concatenated together. All models used the GraphNorm normalization layer and a ReLU as an activation function. Each model was trained for 100 epochs on each fold.

Division of the dataset into training and testing sets was done using cross-validation. The dataset with 7 graphs was divided into one fold with 5 training and 2 testing graphs. Out of all possible combinations of folds, 5 possible folds were randomly chosen for evaluation[1]. To achieve consistent results, the chosen folds are equal for all models.

[1]https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.ShuffleSplit.html

6.1.2 Results

All reported results in this chapter are test set results, unless stated otherwise.

Figure 6.1 shows the mean accuracy of tested models across different folds. Black lines range the weakest and strongest performance on one of the five folds. DeeperGCN outperforms all other models and achieves a mean accuracy of 0.9987. Both models with attention layers performed exceptionally well, achieving a mean accuracy of 0.997. In this experiment there was no difference in performance between the original GAT layer (Veličković et al., 2018) and the newer GATv2 version proposed in (Brody et al., 2022). Results of the multilayer perceptron were significantly lagging behind all graph models, with a mean accuracy of 0.708. This suggests that inclusion of graph structure is crucial for the task. On the other hand, the Graph Isomorphism Model performed the worst out of graph models, with a mean accuracy of 0.914, leading to a conclusion that only topological information is not sufficient for this dataset. Surprisingly, even mere inclusion of isomorphism layers with other convolutional layers led to a drop in performance, which is shown by the *GCN+GIN* and *GAT+GIN* models in Figure 6.1.

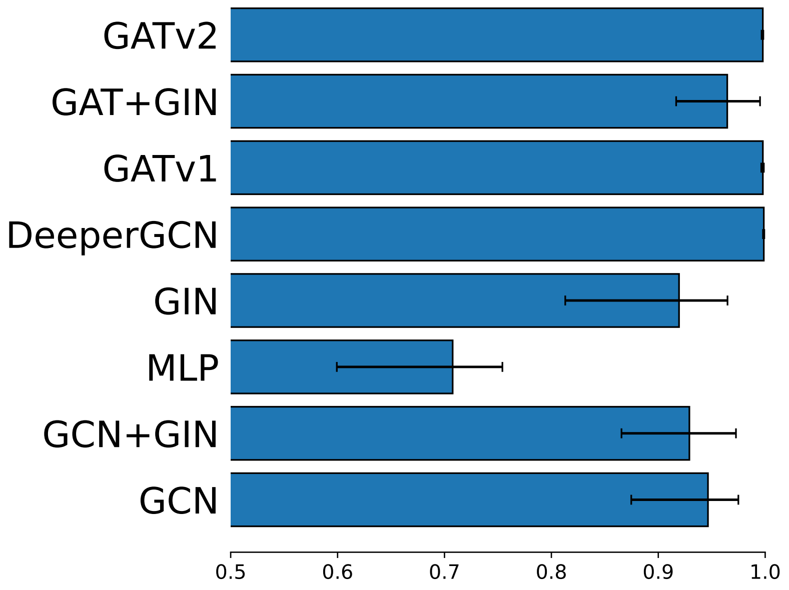


Figure 6.1: Mean accuracy of used models.

To make sure prediction outputs are not biased by the class distribution, a confusion matrix for one of the fold testing sets was created. The confusion matrix in Figure 6.3 confirm that the model is able to predict both classes with exceptional accuracy.

Thanks to the inclusion of graph structure, graph neural networks should be able to generalize well and perform better than classical neural networks with a limited amount of training data. To test this hypothesis, experiments were repeated with a reduced amount of training data. The train test ratio was changed from 5:2 to 3:4. Results in Table 6.1 confirm that the best performing graph neural network (DeeperGCN) is able to achieve comparable

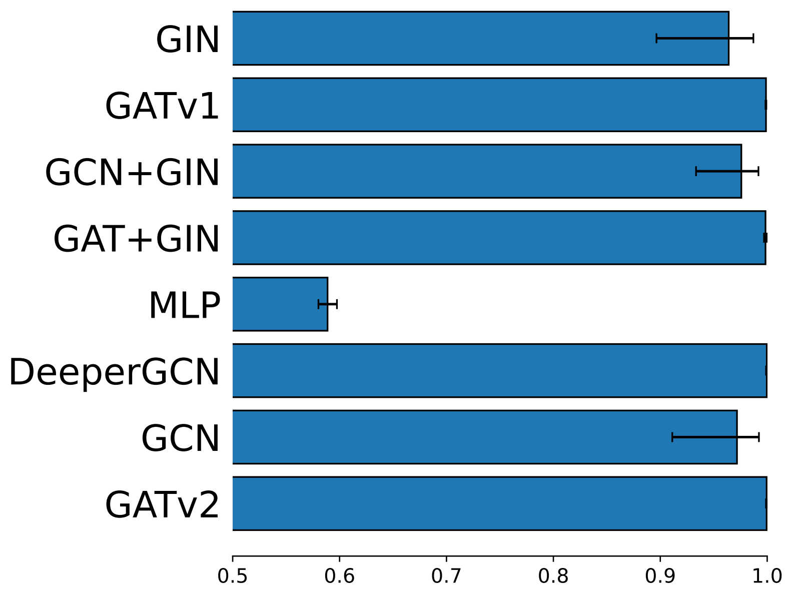


Figure 6.2: Mean AUROC of used models

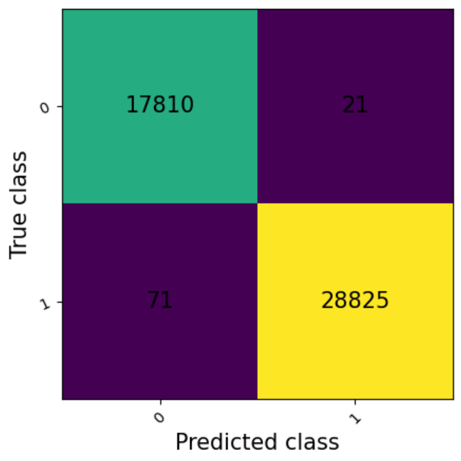


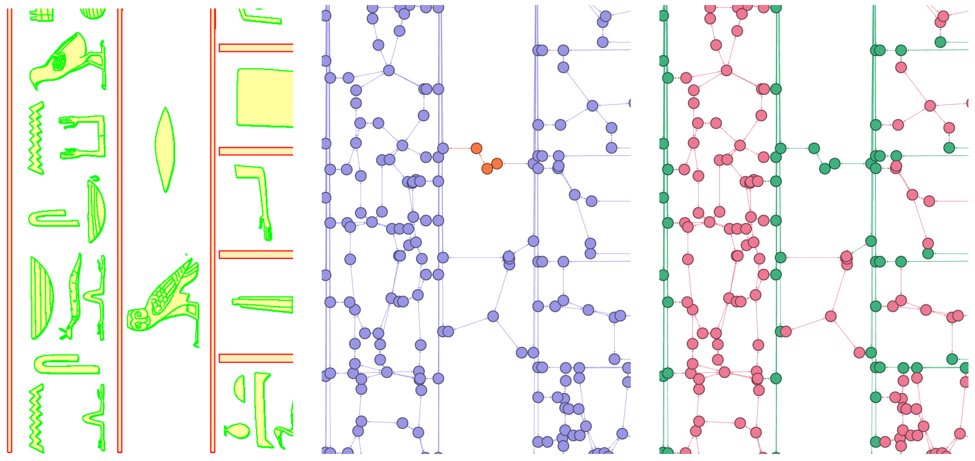
Figure 6.3: Confusion matrix of the best performing model on one of the fold test sets.

results with less training data, while performance of the MLP decreases by 1.5%.

Table 6.1: Mean accuracy with changed train/test ratio.

Model	Mean accuracy 5 train graphs	 3 train graphs	**Difference**
DeeperGCN	**0.9987**	**0.9984**	**-0.0003**
MLP	0.7076	0.6919	-0.0157

Resulting predictions were investigated to find if there are any patterns in wrong classifications. Misclassifications mainly occur at the boundary between horizontally and vertically oriented stripes of hieroglyphs, shown in Figure 6.4.



(a) : Top middle hieroglyph was classified as a separator.

(b) : Wrong node classifications shown in orange.

(c) : Model's predictions, hieroglyphs shown in red.

Figure 6.4: An example of a wrong hieroglyph classification.

6.2 Dynamic graphs

Evaluation of temporal GNNs was done on two tasks. The first task was to predict the proportion of node values, which represent trade values between nations in a graph representing a trading network. It is therefore considered transductive. The second task consisted of a multiclass node classification, with 40 possible classes. Given a citation graph of scientific papers, the goal was to predict the class (field of study) of each node (paper). This task should be considered inductive, because new nodes can appear in the validation and test sets.

6.2.1 Hyperparameters

To facilitate reproducibility of experiments, used hyperparameters are listed below. The batch size was set to 256 unless stated otherwise. The Adam optimizer was used, learning rate was set to 1e-4. Dropout was set to 0.1. Attention layers used 2 heads and 2 attention layers were used if possible. Hidden time dimension was 10. During sampling of neighbors, 10 neighbors were sampled at max. For DyGFormer, number of neighborhood co-occurrence features was 50. The models were trained for 50 epochs.

Static models (GCN and MLP) used a batch size of 8192, learning rate 0.01, dropout 0.5, and number of hidden channels was 256. This setting provided approximately a similar number of neurons to the static and temporal models.

Both models used 3 layers and sampled 10 neighbors at each of the first three-hop neighborhoods.

6.2.2 Results

Table 6.2 shows the validation and test NDCG@10 of the tested models, with columns labeled as reported showing results from the TGB leaderboard[2]. The observed columns show results which were obtained by running the models on my own. For observed runs, only the best results were selected. The majority of results are aligned previously with reported values, which enhances confidence in the validity of the subsequent results. The simple Moving Average baseline outperforms all temporal graph neural networks by a significant margin. TGN is able to outperform the other temporal GNNs on the test set, but results for the validation set are similar for the three selected models.

Table 6.2: Comparisons of model results on the `tgbn-trade` dataset.

	NDCG@10			
	Reported		Observed	
Model	Validation	Test	Validation	Test
TGAT[2]	39.31 ± 0.01	37.40 ± 0.06	0.397	0.381
TGN	0.395 ± 0.002	0.374 ± 0.001	0.394	0.392
DyGFormer	0.408 ± 0.006	0.388 ± 0.006	0.395	0.378
Moving Average	**0.841**	**0.823**	**0.843**	**0.809**

To investigate the claim that performance of temporal GNNs depends on the chosen batch size, all experiments were repeated with three different batch size values. Batch sizes 32, 64 and 200 were chosen. The results in Figure 6.5 show that all models achieved best results with the smallest used batch size, which confirms the theoretical claim based on the design of temporal GNNs. Training with all batch sizes was performed for the same amount of epochs, so that test metrics could be directly confirmed. The results, however, do not accurately reflect the time-information trade-off of selecting a larger batch size, which led to faster training times per epoch. The runtime generally decreased with larger batch sizes, however it was not directly inversely proportional.

After repeating the experiments on the benchmark dataset to ensure expected functionality of the code, temporal GNNs were evaluated on the `ogbn-arxiv` dataset. The results in Table 6.3 show that temporal GNNs underperformed compared even to the simplest static GNN, the GCN. GCN even in an inductive setting outperforms other models. The strongest temporal model is TGAT, which delivers a better result than a multilayer perceptron, which does not use any temporal information.

[2]Uses package version 0.9, some results may vary across versions.

[2]Results of TGAT on `tgbn-trade` are not present on the TGB leadeboards and are taken from the DyGLib paper (Yu, 2023).

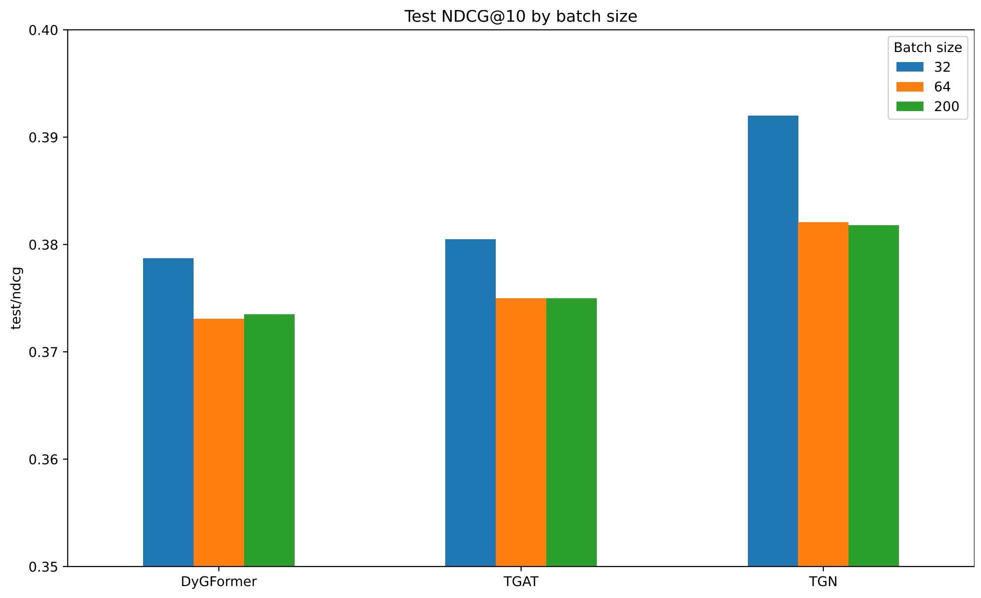


Figure 6.5: Test NDCG@10 on `tgbn-arxiv` across batch sizes.

Table 6.3: Comparison of model results on the `ogbn-arxiv` dataset and its dynamic equivalent. Strongest temporal model results are underlined.

Model	Validation accuracy	Test accuracy
TGAT	<u>0.507</u>	<u>0.487</u>
TGN	0.435	0.380
DyGFormer	0.360	0.401
GCN[1]	**0.578**	**0.536**
MLP[1]	0.502	0.457

Temporal node classification remains to be a very challenging task and although state-of-the-art models were applied on both temporal datasets, more effective architectures of temporal graph neural networks are needed to truly achieve groundbreaking results and overcome static graph neural networks.

[1]The difference between performance of GCN and MLP on the OGB leaderboards and in Table 6.3 is due to the setting. OGB uses transductive learning, while in this experiment I have used static models inductively to be consistent with usage of temporal models.

Chapter 7
Future Work

Although the field of graph neural networks have seen significant advances in the recent years, due to dramatic surge in popularity, there are still many open problems. In terms of implementation, the GNN community would hugely benefit from a library supporting efficient operations for temporal GNNs. This could further accelerate the field of temporal GNNs the way Pytorch Geometric has done to static GNNs. Designing and building such a library would be an interest of mine, but it was outside the scope of this thesis.

Furthermore, the field of temporal GNNs is still in its early days. Most architectures of existing models were designed with temporal link prediction in mind and there is a large space for improvement on temporal node classification tasks. The field of GNNs and temporal GNNs specifically would benefit from broader awareness of the advances made in recent years, which could lead to recognition of new problems where GNNs could be used.

The study of heterogeneous graphs (graphs with multiple types of nodes and edges) is another fascinating area of research. To limit the scope of the thesis, heterogeneous graph methods were not discussed, but heterogeneous GNNs can be applied to knowledge graph analysis, which could provide promising results.

Graph classification was briefly mentioned in the second chapter. Most graph classification datasets are of biological nature, such as predicting characteristics of proteins or discovery of new drugs. Domain knowledge is crucial when designing sound models for these applications. Using GNNs in biology is a logical continuation of the work, because of the inherently graph-like structure.

Chapter 8
Conclusion

In this thesis, I have explored the field of GNNs and focused on static and temporal graph neural networks. I have conducted a comprehensive review of the literature and closely examined several model architectures for both static and temporal GNNs. Static GNNs were applied to a novel real-world application to recognize hieroglyphs based on their graph representation. GNNs have shown impressive improvements compared to traditional neural networks by utilizing the graph topology. The DeeperGCN model achieved near perfect accuracy on the hieroglyph task, and proved to be very efficient in terms of the amount of data needed to achieve solid accuracy.

Examined state-of-the-art temporal GNNs were compared to static GNNs on the node classification task. The benefits and drawbacks of using temporal GNNs and their applications in the broader context were discussed.

The thesis has provided me with an opportunity to explore the field of GNNs, which is a still a seldom studied topic in the artificial intelligence community. I have become familiar with the inner workings of many of the Python deep learning libraries, including Pytorch, Pytorch Geometric, and Lightning. The application of GNNs on the hieroglyph task has guided me through the process of machine learning development, from data preprocessing to model evaluation.

Appendix A
Used Software

In accordance with the Methodological Guideline 05/2023[1], I list the (AI) software used during the creation of this thesis:

- GitHub Copilot[2] for autocompletion of code. Setting to disable suggestions matching public code was used, and only simple one-line completions were accepted.
- ChatGPT[3] for minor reformulations and grammar checks.

[1]https://www.cvut.cz/sites/default/files/content/d1dc93cd-5894-4521-b799-c7e715d3c59e/cs/20230922-metodicky-pokyn-c-52023.pdf

[2]https://github.com/features/copilot

[3]https://chat.openai.com/

Appendix B
Attachments

The following files are attached to this thesis:

```
source_code
├── pixi.toml
└── src
    ├── data
    │   ├── hieroglyphs.py
    │   ├── ogbn_arxiv.py
    │   └── temporal_arxiv.py
    ├── models
    │   ├── base_model.py
    │   ├── deeper_gcn.py
    │   ├── gat.py
    │   ├── gat_gin.py
    │   ├── gcn.py
    │   ├── gin.py
    │   └── mlp.py
    ├── train_arxiv.py
    └── train_hieroglyph.py
hieroglyphs_dataset.zip
```

Appendix C

Bibliography

Järvelin, K., & Kekäläinen, J. (2002). Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, *20*(4), 422–446. https://doi.org/10.1145/582415.582418

Grover, A., & Leskovec, J. (2016). Node2vec: Scalable feature learning for networks.

Kipf, T. (2016). *Graph convolutional networks*. https://tkipf.github.io/graph-convolutional-networks/

Yang, Z., Cohen, W. W., & Salakhutdinov, R. (2016). Revisiting semi-supervised learning with graph embeddings.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry.

Kingma, D. P., & Ba, J. (2017). Adam: A method for stochastic optimization.

Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks.

Hamilton, W. L., Ying, R., & Leskovec, J. (2018). Inductive representation learning on large graphs.

Rossi, A., Tiezzi, M., Dimitri, G. M., Bianchini, M., Maggini, M., & Scarselli, F. (2018). Inductive–transductive learning with graph neural networks. In L. Pancioni, F. Schwenker, & E. Trentin (Eds.), *Artificial neural networks in pattern recognition* (pp. 201–212). Springer International Publishing.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks.

Chiang, W.-L., Liu, X., Si, S., Li, Y., Bengio, S., & Hsieh, C.-J. (2019). Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. https://doi.org/10.1145/3292500.3330925

Falcon, W., & The PyTorch Lightning team. (2019, March). *PyTorch Lightning* (Version 1.4). https://doi.org/10.5281/zenodo.3828935

Fey, M., & Lenssen, J. E. (2019, May). *Fast Graph Representation Learning with PyTorch Geometric*. https://github.com/pyg-team/pytorch_geometric

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks?

da Xu, chuanwei ruan, evren korpeoglu, sushant kumar, & kannan achan. (2020). Inductive representation learning on temporal graphs. *International Conference on Learning Representations (ICLR).*

Li, G., Xiong, C., Thabet, A., & Ghanem, B. (2020). Deepergcn: All you need to train deeper gcns.

Rossi, E., Chamberlain, B., Frasca, F., Eynard, D., Monti, F., & Bronstein, M. (2020). Temporal graph networks for deep learning on dynamic graphs.

Xu, D., Ruan, C., Korpeoglu, E., Kumar, S., & Achan, K. (2020). Inductive representation learning on temporal graphs.

Cai, T., Luo, S., Xu, K., He, D., Liu, T.-Y., & Wang, L. (2021). Graphnorm: A principled approach to accelerating graph neural network training.

Hu, W., Fey, M., Ren, H., Nakata, M., Dong, Y., & Leskovec, J. (2021). Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430.*

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., & Leskovec, J. (2021). Open graph benchmark: Datasets for machine learning on graphs.

Huang, N. T., & Villar, S. (2021). A short tutorial on the weisfeiler-lehman test and its variants. *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* https://doi.org/10.1109/icassp39728.2021.9413523

You, J., Ying, R., & Leskovec, J. (2021). Design space for graph neural networks.

Zeng, H., Zhang, M., Xia, Y., Srivastava, A., Malevich, A., Kannan, R., Prasanna, V., Jin, L., & Chen, R. (2021). Decoupling the depth and scope of graph neural networks. In A. Beygelzimer, Y. Dauphin, P. Liang, & J. W. Vaughan (Eds.), *Advances in neural information processing systems.* https://openreview.net/forum?id=d0MtHWY0NZ

Brody, S., Alon, U., & Yahav, E. (2022). How attentive are graph attention networks?

Kazemi, M. S. (2022). Dynamic graph neural networks. In L. Wu, P. Cui, J. Pei, & L. Zhao (Eds.), *Graph neural networks: Foundations, frontiers, and applications* (pp. 323–349). Springer Singapore.

Leskovec, J. (2022a). *Expressiveness of gnns.* https://web.stanford.edu/class/cs224w/slides/06-theory.pdf

Leskovec, J. (2022b). *Permutation invariance and equivariance.* https://web.stanford.edu/class/cs224w/slides/03-GNN1.pdf

Li, P., & Leskovec, J. (2022). The expressive power of graph neural networks. In L. Wu, P. Cui, J. Pei, & L. Zhao (Eds.), *Graph neural networks: Foundations, frontiers, and applications* (pp. 63–98). Springer Singapore.

Wu, L., Cui, P., Pei, J., & Zhao, L. (2022). *Graph neural networks: Foundations, frontiers, and applications.* Springer Singapore.

You, J., Du, T., & Leskovec, J. (2022). Roland: Graph learning framework for dynamic graphs.

Huang, S., Poursafaei, F., Danovitch, J., Fey, M., Hu, W., Rossi, E., Leskovec, J., Bronstein, M., Rabusseau, G., & Rabbany, R. (2023). Temporal graph benchmark for machine learning on temporal graphs.

Longa, A., Lachi, V., Santin, G., Bianchini, M., Lepri, B., Lio, P., Scarselli, F., & Passerini, A. (2023). Graph neural networks for temporal graphs: State of the art, open challenges, and opportunities.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention is all you need.

Yu, L. (2023). An empirical evaluation of temporal graph benchmark.

Yu, L., Sun, L., Du, B., & Lv, W. (2023). Towards better dynamic graph learning: New architecture and unified library.

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., … Chintala, S. (2024). PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. https://doi.org/10.1145/3620665.3640366

Fey, M., & Lenssen, J. E. (2024). *Advanced mini-batching*. https://pytorch-geometric.readthedocs.io/en/2.5.3/advanced/batching.html

Mařík, R., Landgráfová, R., & Liška, J. (2024). Ancient egyptian hieroglyphic texts structure identification. *Submitted to ACL ML4AL 2024 Workshop*.