Tampere University

Veli-Matti Manninen

LINK PREDICTION IN A TEMPORAL MONEY FLOW NETWORK USING GRAPH NEURAL NETWORKS

Master of Science Thesis
Information Technology and Communication Sciences
November 2022

CONTENTS

ABSTRACT

Veli-Matti Manninen: Link Prediction In A Temporal Money Flow Network Using Graph Neural Networks
Master of Science Thesis
Tampere University
Data Science, Computing Sciences
November 2022

Many real-world phenomenon can be described as a graph that changes over time. Predicting what changes are likely to happen next is a common desire that machine learning pursues to answer. More recently, Graph Neural Networks (GNNs) have emerged as a powerful tool when answering graph related challenges, including link prediction. In this thesis, GNNs are used to study, how the investors of the Helsinki Stock Exchange fund their stock purchases by selling other stocks. This dynamic can be viewed as a flow of money from one stock to another, and can be presented as a network. The goal of this thesis is to establish the base predictability of the money flows for future research with a suitable GNN-based implementation.

The experimental results show that there is predictability in the money flow network that is captured by the methods used. The results are verified against a naive baseline. The findings justify further research on predicting the dynamics of the money flows, and offer themselves as a new baseline for future models.

Keywords: Graph Neural Network, Temporal graph, Link prediction

1. Introduction

Graphs are ubiquitous part of everyday life. Many real-world phenomena can be described as a graph, including road networks, social networks, and the internet. Moreover, graphs are often not static, and instead the connectivity of the graph may change over time. For example, when a road in a road network undergoes maintenance, or an accident happens, the road becomes unusable for traffic. The connection provided by the road therefore ceases for some time period, and resumes when the road is reopened. Similar logic can be assigned to the cables of the internet. In social networks, new friends can be made, and the existing friendships may fluctuate in strength over time. All of the mentioned graphs have a temporal dimension attached to them, but when observed at any given point in time, only a single snapshot of the network is visible. By examining real-world networks that are labeled as static, it quickly becomes apparent, a temporal dimension hides underneath many of them.

Because of their prevalence in describing many useful real-world phenomena, graphs have been studied to a great extent by various disciplinaries. Machine learning as a field is no different. In machine learning, Graph Neural Networks (GNNs) have been proven to be powerful deep learning methods for static graphs. Recently, the focus has shifted to studying GNNs for temporal graphs. In this thesis, the task is to tap into the recent advancements in the research of temporal GNNs, and to perform temporal link prediction on a real-world temporal network dataset of this thesis.

The dataset for this thesis is a weekly binary money flow network between stocks that have been active for trading in the Helsinki Stock Exchange between years 2000 and 2008. The network is observed in weekly snapshots that capture the connectivity of the network at the time of the snapshot. This network is possible to be built based on a unique access to all transactions made by the investors in the Helsinki Stock Exchange. The access is provided by Euroclear Finland.

The definition of the money flow network starts by identifying the stocks that an investor has *net sold* and *net bought* during a time period, here weekly. This means, that during a period of one week, an investor can buy and sell a stock, and only their net is considered. To label the stock as net sold, the investor has sold the stock more in euro amount than bought during that week. Respectively, if an investor has bought a specific stock more

in euro amount, the stock is labeled as net bought for that investor for that week. Each investor gets their own weekly labels for the stocks based on their trading.

The money flows emerge, when the stocks are viewed to be linked from the net sold to the net bought. Each investor has their own money flows based on their net sold and net bought stocks. In this setting, the investor's money can be understood to be "flowing" from the stocks net sold to the stocks net bought. Another way to view this is to think that the stocks net bought are financed by the stocks net sold. The worth of the money flows can be approximated, as was done by Karaila [1] in his network analysis of the Finnish financial markets during the financial crisis of 2008. However, in this thesis, the money flows are treated as binary, where a money flow either exists or it does not. For a money flow to exist, *any* amount of money flow from one stock to another is sufficient. Additionally, money flows that enter or exit the stock market are ignored. For example, if an investor buys stocks but does not sell stocks, money enters the stock market, and is thus ignored. Similarly, if an investor sells stocks but does not buy stocks, the money exits the stock market.

To represent the money flows on the level of the whole stock market, the investors' money flows are aggregated. An aggregated weekly money flow from one stock to another is likewise binary. For it to exist, it is sufficient to know that there exists at least one investor that has a money flow between the said stocks. In other words, it is sufficient to know that at least one investor has net sold and net bought the said stocks. The weekly aggregated money flows can then be represented as a directed network, where the stocks are nodes and the money flows are links. This network constructed by the aggregated weekly money flows represents one snapshot of the money flow network of the Helsinki Stock Exchange. Each week gets its own snapshot network, and the dataset of this thesis is the sequence of those weekly snapshots.

Separate from the weekly snapshots, weekly features for the stocks are observed. These features include: revenue, number of days the market has been open, shares outstanding, volatility, volume, and the number of trades.

Other than the research done by Karaila [1], the author of this thesis is not aware of any other research made on money flows between stocks or temporal link prediction between them. Therefore, the contribution of this thesis is to establish the base predictability of the money flow network for future research, using binary money flows with a suitable GNN-based method.

Since the money flow network is observed as a series of weekly snapshots, the goal is to predict the connectivity of the money flow network at time $t + 1$, where t is the time step of the last observed snapshot. Because the graph is directed, predicting merely the existence of a connection between two stocks is not sufficient, and instead the direction of the connection must also be predicted. This is an extra challenge, especially because

the research on directed temporal GNNs is understudied.

The methodology used for predicting the next snapshot is tackled with an autoencoder [2] architecture along with the following methods: a Graph Attention Network (GAT) [3], a Gated Recurrent Unit (GRU) [4], and a reconstruction neural network. The division of labor between the methods is such that GAT learns the local dynamics in the snapshots the same way as it would on a static graph. GRU is then used to differentiate the learned local dynamics produced by the GAT, in order to mine the temporal dynamics between the snapshots into a new temporal representation of the graph. GAT is chosen to model the structural dependency because it is not bound to model similarity between stocks, unlike many other alternatives. Instead, GAT has the ability to model more complex relationships between stocks, such as repulsion. GRU is chosen because of its ability to efficiently model sequential data, which the sequence of graph snapshots essentially is.

An autoencoder architecture arises, when the model learns a new representation for the input data, which is reconstructed back into the shape of the input [2]. Here, GAT and GRU learn a representation from the snapshots, and this representation is reconstructed into the new prediction snapshot with a neural network. The autoencoder architecture is chosen for this work because it permits the model to learn the temporal representation of the money flow network, and to solve the added challenge of predicting the direction of the links.

The chosen model should be able to perform better than a naive baseline prediction. As for the baseline, the current graph snapshot acts as the prediction for the next snapshot. The baseline is denoted as $H_0 : G_t = G_{t+1}$.

In total, 3 research questions (**RQ**) are introduced for this thesis:

RQ1: Does the model outperform the baseline?

RQ2: Do the available input features improve the performance of the model?

RQ3: Does modeling the temporal evolution improve the performance of the model

The rest of the thesis is organized as follows. Chapter 2 introduces graphs and a mathematical definition for the binary money flow network. Chapter 3 introduces machine learning, GNNs, and link prediction. Chapter 4 presents the descriptive statistics of the dataset, dissects the methodology used, and covers the experimental setup for the link prediction task. Chapter 5 presents the results of the link prediction task. Chapter 6 provides suggestions for future research. Chapter 7 draws conclusion from the contents of this thesis.

2. Graphs And Networks

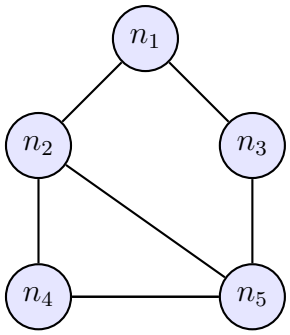
A graph is a set of objects that can have relationships between each other. The objects are called nodes, and the relationships are called links. A graph is not to be confused with the graph of a function or to a data visualization chart. Instead, a graph can simply be understood as a synonym to a network. Mathematicians typically tend to discuss in terms of graphs when talking about arbitrary and theoretical concepts of relationships between objects. A network on the other hand typically is a real-world phenomenon, such as a social network, road network or the internet. Further arguments, how a graph and a network differ from each other can be made, but for all practical purposes they can be viewed as synonyms to each other. However, the established naming conventions are honored in this work, and graphs are mostly discussed in theoretical settings and networks with real-world phenomenons.

A static graph can be denoted as $G = \{V, E\}$, where the nodes are $V = \{1,...,n\}$ and the links are $E \subseteq V \times V$. A graph is said to be static if the contents of V and E do not change. This is in contrast to temporal graphs, where changes in the set of nodes and links are allowed. Temporal graphs are discussed in more detail later.

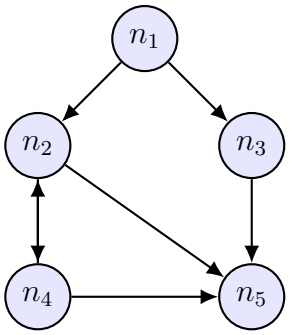
A link is defined between a source node i and a destination node j. This definition implies, that while a node j is connected to node i with a link l_{ij}, it does not mean that node i is connected node j. For this to happen, there needs to exist another link l_{ji} that connects j to i. Therefore, to connect a pair of nodes bidirectionally, two links are required.

A graph is said to be undirected if every connection between two nodes is bidirectional. Conversely, if there exists at least one connection between a pair nodes that is not bidirectional, the graph is called directed. The World Wide Web is an example of a directed network, where a website can have a link to another website, but not necessarily the other way around. A network of friends is an example of an undirected network, where two people either are friends or they are not. The strength of the friendship may vary, but the connection must be bidirectional by definition for friendship to exist.

The number of connections a node has in an undirected network is called the degree of the node. For directed graphs, the degree can further be divided to in- and out-degree by calculating the incoming and outgoing connections respectively.



(a) Undirected



(b) Directed

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2.1. *Graph (a) is undirected and (b) is directed. The corresponding adjacency matrices are below.*

2.1 Graph Representations

A typical way of representing graph is by matrices. A matrix that encodes the connectivity between the nodes is called an adjacency matrix. In an adjacency matrix, each row and column represents a node. The value of the element at row i and column j determines, if the two nodes have a link between them. In other words, if the nodes are adjacent.

Typically the ordering of the nodes is assumed to be the same in both row and column axis. An example of an undirected and directed graph is illustrated both visually and with an adjacency matrix representation in Figure 2.1.

If the value of the link in the adjacency matrix is 0, the link is considered not existing. The values of the adjacency matrix do not need to be binary, as is illustrated in Figure 2.2. Rather, the values in the adjacency matrix can be viewed as the weight of the link, where non-zero values indicate that the link exists with the said value as its weight.

It is possible for a node to be connected to itself, creating a so called self-loop. Self-loops appear as diagonal elements in the adjacency matrix, where the row index i is equal to column index j. In the previously mentioned examples, there are no self-loops and therefore, every diagonal element is zero.

The adjacency matrix of an undirected graph is symmetric, meaning, the values of both sides of the diagonal are mirror images of each other. This is due to the fact that every link in an undirected graph is bidirectional. The adjacency matrix of a directed graph is asymmetric, because not every link is guaranteed to be bidirectional. This creates

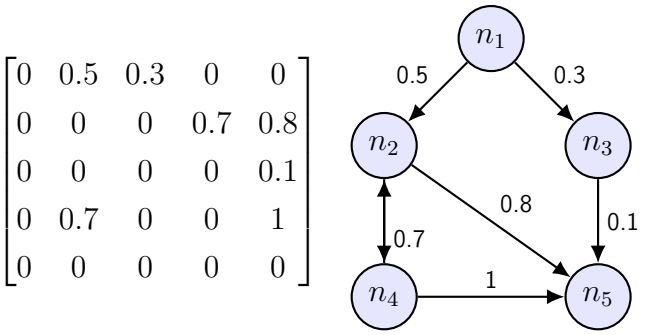


Figure 2.2. *Example of a weighted graph. The weights of the links can be placed directly into the adjacency matrix.*

the need to define, which axis of the matrix represents the source nodes and which the destination nodes. This convention varies within disciplines [5, 6]. In this thesis, the directed adjacency matrix is defined as the row-axis representing the source nodes and the column-axis representing the destination nodes.

2.2 Temporal Graphs

A temporal graph (also dynamic graph) is a graph that changes over time. The changes can occur in the structure, meaning the number of the nodes or the connectivity between the nodes, or in the contents of the nodes or links. When a change occurs, the time of the change is associated with the change. Based on how the time is recorded, the temporal graphs can be divided into two categories: *continuous* and *discrete* temporal graphs [7].

Continuous temporal graphs can change at any given time. Many real-world networks, such as social networks, can be viewed to be continuous temporal graphs, where every change can be individually recorded and attached with a timestamp. Sometimes the underlying continuous temporal graph is not possible to be observed in continuous time, and is therefore limited to be observed in discrete time. This is called discretization of the temporal graph. In fact, it is often the case that discrete temporal graphs arise as a result of discretization of the continuous counterpart, instead of them occurring naturally in discrete time domain.

It should be noted, that if a continuous temporal graph is discretizated, some temporal information is lost. It is not possible recover the exact temporal timestamps for the changes when they happened in the graph between two discrete observations. Instead, every change will receive a timestamp when it was first observed.

In simplistic terms, continuous temporal graph is a sequence of individual graph changes, while a discrete temporal graph is a sequence of graph snapshots. Interest in continuous temporal graphs is increasing, and they are the subject of active research [8]. However,

the focus in this thesis is on discrete temporal graphs.

2.3 Money Flow Network

The money flow network can be represented as a discrete temporal graph that is observed as a sequence of snapshots, denoted as:

$$G = \{G_1, G_2, ..., G_t\} \tag{2.1}$$

where G_t is the graph snapshot that represents the aggregated money flows of the investors at time t.

To define the money flows for the investors, the stocks first need to be labeled, whether they have been net sold or net bought:

$$s_i^k(t) : \sum S_i^k(t) > \sum B_i^k(t) \tag{2.2}$$

$$b_i^k(t) : \sum B_i^k(t) > \sum S_i^k(t) \tag{2.3}$$

where $s_i^k(t)$ and $b_i^k(t)$ are the respective truth labels for net sold and net bought for stock i by investor k in time period t, $S_i^k(t)$ is the set of sales and $B_i^k(t)$ is the set of purchases of the stock i by investor k in time period t in euro amounts. By convention, the sum of an empty set if zero.

If the truth labels are treated as binaries, where $True = 1$ and $False = 0$, the existence of an investor's money flow from stock i to stock j in time period t can be expressed as:

$$l_{ij}^k(t) = s_i^k(t) b_j^k(t) \tag{2.4}$$

This definition implies that any amount of money flow is sufficient for the money flow to exist. To aggregate the investors' money flows to represent the existence of a money flow on the level of the whole stock market in time period t, finding only one investor with the money flow from stock i to stock j is sufficient:

$$l_{ij}^t : \exists k \in D_t : M(k, t, i, j) \tag{2.5}$$

where D_t is the set of investors at time t, k is an investor, $M(k, t, i, j)$ means the investor k has net sold the stock i and net bought stock j in the time period t.

Again, if the truth values are treated as binaries, l_{ij}^t directly implies, whether the money

$$G_t = \begin{bmatrix} l_{11}^t & \cdots & l_{1N}^t \\ \vdots & \ddots & \vdots \\ l_{N1}^t & \cdots & l_{NN}^t \end{bmatrix}$$

Figure 2.3. *A snapshot of the money flow network at time t represented as an adjacency matrix. N is the number of stocks in the network.*

flow exists on the aggregate level. The aggregated money flows may be placed in an adjacency matrix, as in Figure 2.3, to represent the money flows in the snapshot G_t as a directed binary network.

2.4 Isomorphism, Permutation Invariance and Equivariance

The nodes in a graph have no canonical order between them. For example, even though Figure 2.4 shows two different visual graph representations, the underlying structure is identical in both of them. Every node has the same adjacent nodes, and only the order of the nodes on the 2D plane has changed. This property is called isomorphism.

Representing isomorphic graphs by adjacency matrices becomes with a challenge. A single graph can be represented by multiple different adjacency matrix forms, yet the underlying graph is the same. For example, in Figure 2.5, the graph is represented with two different adjacency matrices, both of which encode the same connectivity as the visual graph depicted on their left side. Note the order of the nodes in the adjacency matrices is not equal. This is emphasized with row and column headers.

In fact, every graph can be encoded by $n!$ isomorphic adjacency matrices, where n is the number of nodes, and ! is the symbol for factorial. The graph in Figure 2.5 can be represented, not by two, but by $5!$ different adjacency matrices, resulting in a total number of 120 different adjacency matrices. The connectivity remains the same in the adjacency matrices because the links are merely permuted correspondingly. They key

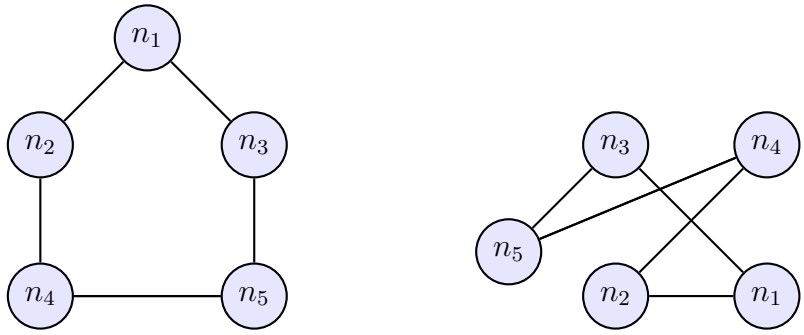
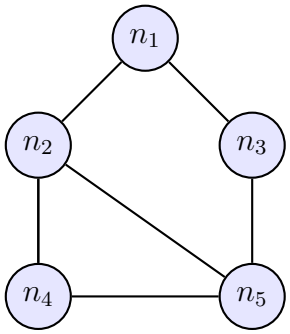


Figure 2.4. *Two isomorphic graphs embedded arbitrarily on a 2D plane.*



$$\begin{array}{c} \\ n_1 \\ n_2 \\ n_3 \\ n_4 \\ n_5 \end{array} \begin{array}{c} \begin{array}{ccccc} n_1 & n_2 & n_3 & n_4 & n_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ n_2 \\ n_1 \\ n_5 \\ n_4 \\ n_3 \end{array} \begin{array}{c} \begin{array}{ccccc} n_2 & n_1 & n_5 & n_4 & n_3 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2.5. *An undirected network with two possible adjacency matrix representations out of many.*

point in the context of this thesis is that if any adjacency matrix were to be fed into a function, the output of the function should be the same, because the underlying graph is the same [9]. In other words, the function must be agnostic, or invariant, of the permutation of the nodes in the adjacency matrix.

Linear algebra is a branch of mathematics, where matrices play an integral role. Consequently, it provides a mathematical definition for permuting the rows and columns of a matrix. A single permutation of both the the rows and columns of the adjacency matrix is done by a matrix multiplication: $\mathbf{PAP}^\top$, where $\mathbf{A}$ is the adjacency matrix, $\mathbf{P}$ is a permutation matrix, and $\mathbf{P}^\top$ is the transpose of $\mathbf{P}$. The result of this matrix multiplication is an adjacency matrix, where both the rows and columns are correspondingly permuted according to the permutation matrix $\mathbf{P}$.

In order to design a permutation invariant function $f(\cdot)$ that takes the adjacency matrix $\mathbf{A}$ as input, it must be of form:

$$f(\mathbf{PAP}^\top) = f(\mathbf{A}) \tag{2.6}$$

Additionally, other information regarding the nodes, called node features, are often desired to be included as input to the function. These node features are often in matrix form, and this feature matrix also needs to permuted with the same permutation matrix:

$$f(\mathbf{PAP}^\top, \mathbf{PX}) = f(\mathbf{A}, \mathbf{X}) \tag{2.7}$$

where $\mathbf{X}$ is the node feature matrix of form $\mathbb{R}^{N \times F}$, N being the number of nodes, and F is the number of node features.

It should be noted, that a permutation invariant function will destroy the information about which permutation was used on the input. Sometimes this is a desirable property and sometimes not. When it is important that the function preserves the order of the

input, permutation invariant functions are not applicable. In contrast to a permutation invariant function, a function that preserves the order of the input is called a permutation *equivariant* function, and can be denoted as:

$$f(\mathbf{P}\mathbf{A}\mathbf{P}^{\top}, \mathbf{P}\mathbf{X}) = \mathbf{P}f(\mathbf{A}, \mathbf{X}) \tag{2.8}$$

In other terms, the output of a permutation equivariant function is the same regardless if the permutation matrix is applied on the input or the output.

3. Machine Learning

Machine learning is a study of algorithms that are able to learn from data. Often a practical goal of machine learning is to teach machines abilities to solve tasks that: humans are unable to perform, are too laborious to perform, or to improve the cost benefit to perform.

This chapter covers the basics of machine learning in order to understand, how graph neural networks work, and how they may be utilized to perform link prediction on the dataset of this thesis. While link prediction has utilization outside of machine learning, since it is performed with machine learning methods in this thesis, it is presented under this chapter.

3.1 Temporal Link Prediction

Link prediction is a task of predicting the existence of a link between two nodes in a graph. The predictions can be for new links, or for discovering unobserved links in a partially observed graph.

Temporal link prediction differs from the traditional link prediction by taking into account the removal of links [10]. This is important because in many real world networks the dynamicity stems from reappearing links [10]. A link might be removed at some point and appear again at later time. This requires the link prediction method to produce predictions not only for new links but also for currently existing links because they might be removed [10]. Respectively, the method also has to accurately predict non-links to remain non-links.

In temporal networks, the links are not independent and identically distributed (iid) [10, 11]. Yang et al. [11] showed that the geodesic distance between nodes strongly influences the probability of forming a link. Nodes with shorter geodesic distance to each other are more likely to have a link. Geodesic distance between two nodes in a binary network is defined as the fewest number of links connecting a pair of nodes.

Junuthula et al. [10] expanded this idea to temporal networks and observed that links that have previously existed are much more likely to reappear in the future, compared to links that have never existed. In their work, they treated the geodesic distance of two

		Ground truth	
		Positive	Negative
Prediction	Positive	TP	FP
	Negative	FN	TN

Table 3.1. *Confusion matrix.*

previously connected nodes as 1. The phenomenon of classifying links to previously seen and never seen links is amplified in sparse networks, where the proportion of unseen links is typically more pronounced. These observations also have implications on how the link prediction method should be evaluated.

Typically the link prediction method outputs a score matrix $\mathbf{S}$, where $\mathbf{S}_{ij}$ is the real valued score between [0, 1]. The scores represent the predictor's confidence for the link $\mathbf{S}_{ij}$ to exist. In order to compare the results to the ground truth, the scores are converted to ones and zeros based on a threshold. Above or at the threshold, the score is converted to 1, otherwise to 0. The converted values are then compared to the ground truth binary values. Based on the comparison, the predictions can be divided into 4 categories: true positive (TP), false positive (FP), true negative (TN), and false negative (FN). These categories can be summarized as a confusion matrix, depicted in Table 3.1. In link prediction, the positive class means the existence of a link.

By calculating different ratios between the 4 categories, various metrics can be created. In this thesis, the most important metrics to consider are: true positive rate (TPR), false positive rate (FPR), positive predictive value (PPV). Below are their respective equations and their often used aliases:

$$\text{True Positive Rate (TPR/recall/sensitivity)} = \frac{TP}{TP + FN} \tag{3.1}$$

$$\text{False Positive Rate (FPR)} = \frac{FP}{FP + TN} \tag{3.2}$$

$$\text{Positive Predictive Value (PPV/precision)} = \frac{TP}{TP + FP} \tag{3.3}$$

The metrics depend on what threshold was used to compute the confusion matrix. By choosing a different threshold, the contents of the confusion matrix may be different, and thus the ratios would also be different. Selecting a fixed threshold is difficult, and instead a sliding threshold between [0, 1] is used to compute every possible unique confusion matrix. The metrics are then calculated for each unique confusion matrix.

In binary classification, which link prediction often is, common metrics for evaluation are

true positive rate and false positive rate. Creating the metrics with a sliding threshold and comparing them against each other results in the Receiver Operating Characteristic (ROC) curve. Calculating the area under the curve is a common method to summarize the performance of the link prediction method. The area under the ROC curve is often abbreviated to AUROC.

The problem with AUROC in temporal link prediction is that it tends to over emphasize the easier problem of predicting previously seen links [10]. Predicting solely zeros for the harder problem (new links) has the effect of inflating the evaluation score of AUROC, while simultaneously obfuscating the model's performance on predicting new links. This is because an unseen link has a lower probability of appearing, and often times most of the unseen links will remain unseen. Therefore, predicting unseen links to remain unseen produces a lot of correct true negative classifications, which will in turn inflate the AUROC score, making it a biased evaluation metric.

The inflation caused by the true negatives can be bypassed by using a different metric. Comparing TPR and PPV against each other will result in the Precision Recall (PR) curve. Calculating the area under the PR curve (PRAUC) is used to evaluate how well the model performs exclusively on the positive class. However, in reality, predicting when a link does not exist is still a relevant task in temporal networks and cannot simply be ignored [10].

As a solution, Junuthula et al. suggest evaluating the problems of predicting previous and new links separately. For evaluating the previously seen links, where predicting both the negative and positive classes are relevant, Junuthula et al. suggest using the AUROC. For the new links, where the focus is predicting the occurrence of rare new links, while ignoring the prevalent negative class, Junuthula et al. suggest using PRAUC.

3.2 Supervised Learning

Supervised learning is a circumstance in machine learning, where the dataset provides both the input data and the labels for the data [12]. A label is the correct output value for the corresponding item in the dataset. For example, in image recognition, if the task is to determine the existence of a cancer tumor in x-ray images, the image itself would be the data item, and the information whether the image contains a tumor is the label.

The learning is "supervised" by letting the model make label predictions for the data items, after which the true labels are shown to the model. The learning happens, when the model adjusts its reasoning on how it makes the prediction, based on how wrong the prediction is compared to the true label.

Furthermore, the goal is to teach the model to make generalized predictions for all possible inputs, not just for the seen data [12]. In the x-ray image example, the goal

would be to teach the model to classify future, unseen x-rays correctly, and not merely the existing ones.

The advantage of supervised learning is that the labels offer a natural way to harness the information within the data. They also offer clarity how to approach the problem. By defining the labels, humans have control over what is being modeled and how to measure the progress. The disadvantage is that the model can only learn what it is told to and cannot discover unknown information [12].

The ultimate obstacle in supervised learning is that sometimes the labels are not available, or are only available for a portion of data. In these cases supervised learning is not applicable and other machine learning paradigms need to be used.

3.3 Artificial Neural Network (ANN)

An Artificial Neural Network (ANN) is a paradigm in machine learning, that attempts to loosely imitate the biological brain. The motivation behind the ANN is to try to harness the complex problem solving capabilities the brain has by emulating the learning process of the brain. Often artificial neural networks are simply called Neural Networks (NN), if the concept is within machine learning.

In brains, the neurons are connected to other neurons by synapses. If a neuron receives a strong enough signal through its synapses, they become activated and fire an output signal. The output signal is then forwarded to all neighboring neurons via a connecting synapse. These neurons will then treat the arriving signal as their input signal, and the procedure repeats. Multiple neurons interconnected this way form a neural network. This biological concept can be translated into mathematics, and subsequently for the use of machine learning, through the lens of matrices, activation functions, and differentiation.

In an ANN, each connection to an artificial neuron is accompanied by a weight. The input that travels via the connection is then multiplied with the corresponding weight, producing a weighted signal. The inputs to the neuron can be the original inputs for the ANN, or the outputs from the other connected neurons. The sum of the weighted signals is then inserted into an activation function, which determines the output of the neuron.

Typically the artificial neurons are arranged in layers, where the neurons in a layer are exclusively connected to the neurons in the preceding layer and to the neurons in the following layer. Thus, the output of the neurons in the previous layer is the input of the neurons in the next layer. This setup is referred as a feedforward neural network (FNN), because the passage of information travels only forward. If there exists more than 2 layers, the layers between the first and last are called hidden layers. An example of a feedforward network is illustrated in Figure 3.1. Other architectures exist, such as

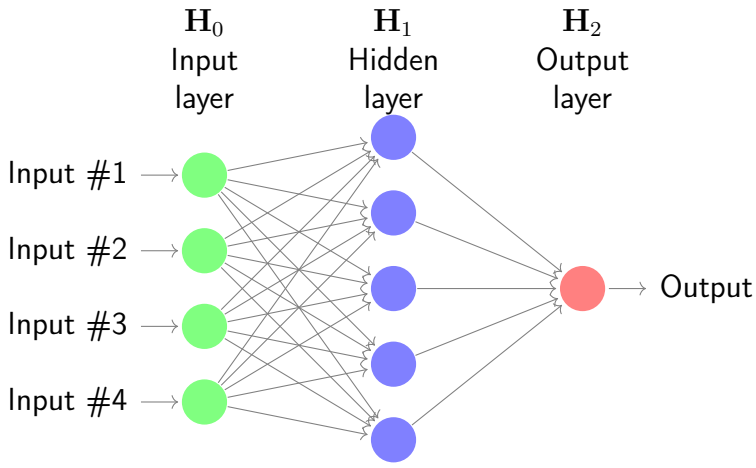


Figure 3.1. *An example of a feedforward network [13], where k = 2. The number of hidden layers and neurons is arbitrary. Only one hidden layer containing 5 hidden neurons is used here. The number of output neurons is task dependent, here is a single output neuron.*

a Recurrent Neural Network (RNN) [14], where neurons can be connected within the layer, or to the previous layers. The output of a layer in a feedforward network can be expressed mathematically with matrices:

$$\mathbf{H}_k = \sigma\left(\mathbf{W}_k \mathbf{H}_{k-1}\right) \tag{3.4}$$

where k is the layer number, where $k = 0$ is the input layer, $\mathbf{W}_k$ is the weight matrix of layer k, $\mathbf{H}_{k-1}$ is the output matrix of the previous the layer, σ is a non-linear activation function, and $\mathbf{H}_k$ is the output matrix of the layer k.

Supervised learning with an ANN is achieved by comparing the final output of the ANN to the expected result, finding their difference, and minimizing the difference by adjusting the weights appropriately. The function that calculates the difference is commonly called the cost, error, or loss function. To know how to adjust the weights, the loss function is differentiated with respect to the current weights. The differentiation reveals whether to increase or decrease the weights, in order to reach a local minimum of the function. The weights are then updated slightly towards that direction, that minimizes the function. After the update, the learning procedure repeats until the output of the loss function no longer decreases.

The process of minimizing the loss function is not guaranteed to perfectly learn the problem, because differentiation can only find local minimums of the loss function, based on the initial weights. There exists various different strategies for updating the weights and they are all commonly referred as optimizers.

3.4 Feature Learning

The data in a dataset can be viewed to be represented by variables and their realisations. In machine learning, the variables are typically called features [15].

The original features are often not suitable as an input for machine learning algorithms, and therefore need to be converted to a more suitable form [15]. In addition, many machine learning tasks benefit, if the features are as informant, discriminant and independent as possible [16]. Humans can craft relevant features based on their natural cognitive intuition and through previously gathered domain expertise. While it has been shown that this kind of feature engineering is important, the process is often very time consuming [15].

Feature learning is a concept, where a system, such as a neural network, tries to learn how to convert the input data into relevant features for the machine learning task [16]. The features learned by a machine might not be obvious for humans, or they might be completely unreachable due to the amount of labour needed. At the same time they perform as well or better as human designed features [16].

Feature learning is also called representation learning [16]. The process of converting the input features into new learned features can be viewed as mapping of the original feature space into a new vector space. The new vector space is called latent or hidden space, because its not directly accessible for humans. The naming of the new vector space and the new representations vary because feature learning techniques have independently developed from different branches of mathematics and machine learning. In this thesis, the new representations of the data in the latent vector space are called latent/hidden representations/features, or embeddings.

3.5 Deep (Feature) Learning

The concept of deep learning roots from the idea, that by performing feature learning again on the latent features, machines can learn more abstract and complex latent features [17]. After applying feature learning on the input data, the resulting latent features are fed as an input to another feature learning operation. This in turn outputs new latent features, which in turn can be fed to yet another feature learning operation. This hierarchical process can go even further, or "deeper", arbitrarily many times, provoking the notion of deep (feature) learning.

One of the most successful neural network in deep learning is the Convolutional Neural Network (CNN) [18]. However, the problem with CNN, and deep learning in general, is that the success has been achieved by operating on Euclidean domain. Graphs by contrast are generally non-Euclidean. CNN can only operate on regular Euclidean graphs, such

as text (1D chain-graph) and images (2D grid-graph), making it an invalid choice for arbitrarily shaped graphs [19].

Recent interest in research has been to find ways to extend deep learning to non-Euclidean domains, such as graphs. Many different methods have emerged independently from different disciplines. In order to unify these methods under an umbrella term, Bronstein et al. [9, 20] propose the term *geometric deep learning* for those deep learning methods that generalize to non-Euclidean domains.

3.6 Encoder & Decoder

Latent features produced by feature learning are not themselves that meaningful. They can often times be viewed as an intermediate stage of the deep learning process. In order to obtain tangible results from the latent representations, the information encoded in the latent representation need to be decoded into something meaningful. What is meaningful is in turn determined by the task at hand. For example, when the task is to produce captions for images, the information encoded in the latent features need to be somehow translated into text that represents, what is depicted in the image [21]. The function that does that is called a decoder. Respectively, the function that creates the latents is called an encoder. Both working together form the encoder-decoder pair.

The encoder-decoder pair operates such that the encoder receives the original input and creates latents that encode useful information from the input. The decoder then receives the latents as its own input and outputs a meaningful and task-dependant answer to the problem. The decoder can be a learnable function, which is trained simultaneously with the encoder. This all depends on the task in hand and on the architecture of the model.

A special case of the encoder-decoder pair is the autoencoder [22, 23]. The output of the decoder in the autoencoder is as close as possible to the shape of the input of the encoder [23]. For example, if an autoencoder receives an image as an input, the output should also be an image. Respectively, if the input is a graph, the output should also be a graph.

Autoencoders have traditionally been used as a dimensionality reduction technique, since the latent features in the autoencoder act as a sort of an information bottleneck [23]. An information bottleneck is achieved by setting the output dimension of the encoder low. For an autoencoder to be able to decode a meaningful output from the low dimensional latent representation, the unnecessary information has to dissipate. This makes room for the most relevant information to be condensed in the latent representation.

After the decoding, the dimensionality of the original input can viewed to have been reduced to the size of the latent space, because the output could be decoded from the low latent space alone. This makes it possible to discard the original input data and

keep only the low dimensional latent features. However, the dimensionality of the latent features does not have to be low, and instead can be set arbitrarily. This means that the autoencoders are not bound to perform dimensionality reduction.

3.7 Graph Neural Networks (GNNs)

A Graph Neural Network (GNN) is a neural network that takes a graph as an input. GNNs have been around for almost two decades [24, 25] but in recent years they have garnered particular interest. Zhou et al. [19] attribute the resurgence of GNNs to the following motivations.

The first motivation is to benefit from the success of deep learning, but for graph-structured data [19]. As stated earlier, deep learning has earned its renown by operating on Euclidean domain, whereas graphs are typically non-Euclidean.

The second motivation is to find better solutions for graph feature learning [19]. Feature learning has previously been applied on graphs with so called shallow encoders, such as node2vec [26] and DeepWalk [27]. However, these methods lack some important properties that a desirable feature learning algorithm would possess when dealing with graphs. The limitations of shallow encoders include: their inability to scale efficiently to large graphs, they cannot be used for dynamic graphs, they cannot generalize to unseen nodes, and they cannot take into account node features [28, 29]. GNNs are free of these limitations [28, 29].

GNNs are currently experiencing a lot of attention because of their ability to produce great results in many graph related tasks [30]. As a result, they are an actively researched topic in machine learning [31].

The rest of this chapter will cover how the GNN learns latent node representations by aggregating the node neighborhoods, how the node representations can be decoded for common down stream tasks, and how GNNs can be utilized with temporal graphs.

3.7.1 Neighborhood Aggregation

The neighborhood of node i consists all the nodes adjacent to node i and is denoted as $\mathcal{N}(i)$. The latent representation of node i is created by feeding the features of the nodes in the neighborhood $\mathcal{N}(i)$ into some permutation invariant aggregation function, such as sum, max or mean. The result of the aggregation is then ran through some non-linear function, resulting to the new latent representation of node i.

If the process is repeated, with the latent features in place of the original features, the node i will learn information from nodes outside of its neighborhood. Namely, information about the neighbors of the neighbors. This is because in the previous iteration, every

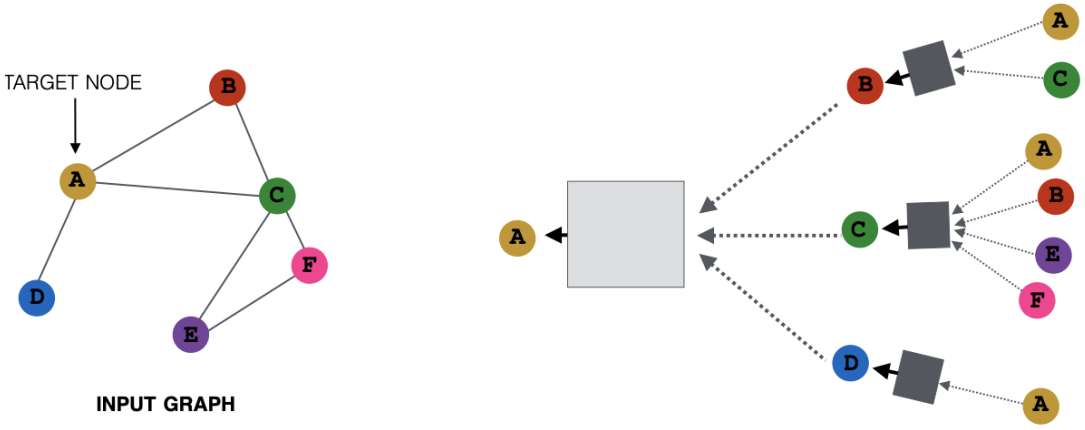


Figure 3.2. *Input graph (left) and the computation graph for 2-hop neighborhood aggregation of node A (right). [28]*

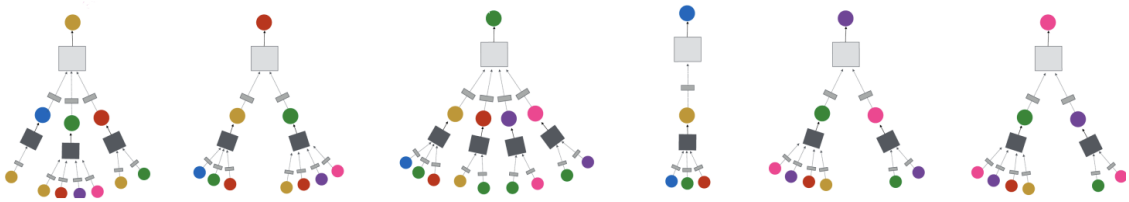


Figure 3.3. *Computation graphs for 2-hop neighborhoods for all nodes in the graph. [32]*

other node has performed its own neighborhood aggregation, and encoded information about their own neighborhood in their own latents. When the nodes perform neighborhood aggregation again, the nodes will indirectly learn about nodes further away, even though each node only ever aggregates its own neighborhood.

If the process is repeated k times, the nodes will receive information from nodes k-distance away. All of the nodes reached within this radius are called the k-hop neighborhood. This setup is analogous to layers in a neural network, where k is the number of layers.

The process of neighborhood aggregation is illustrated in Figure 3.2, where the neighborhood aggregation is performed on node A when $k = 2$. The graph on the right is the computation graph of A, and it shows, how the latent feature is calculated from the features of the neighbors. The black and grey boxes represent the layer in a neural network. Respectively, each other node in the graph will create their own computation graph, which is illustrated in Figure 3.3.

More formally, a neighborhood aggregation of a node can be denoted as:

$$\mathbf{h}_i^{k+1} = \sigma\left(\bigoplus_{j \in N(i)} \mathbf{h}_j^k\right) \tag{3.5}$$

where $\mathbf{h}_i$ is the latent feature of the node i, k is the number of hops, $\oplus$ is a permutation invariant aggregation function, $N(i)$ is the neighborhood of node i, and σ is a non-linear function. If the graph is directed, it must be specified, whether the neighborhood consists of the incoming or outgoing nodes.

After applying a permutation invariant function to every node's neighborhood, the output of the GNN is a permutation equivariant latent feature matrix. This ensures that the latent features can be identified to be belonging to a certain node, because the order of the nodes persists in the output matrix. Therefore, the function of GNN can be expressed as:

$$f(\mathbf{PAP}^T, \mathbf{PX}) = \mathbf{P}\begin{bmatrix} g(\mathbf{x}_1, \mathbf{X}_{\mathcal{N}(1)}) \\ g(\mathbf{x}_2, \mathbf{X}_{\mathcal{N}(2)}) \\ \vdots \\ g(\mathbf{x}_i, \mathbf{X}_{\mathcal{N}(i)}) \end{bmatrix} = \mathbf{PH} \tag{3.6}$$

where $f(\cdot, \cdot)$ is the function of the GNN, $\mathbf{P}$ is a permutation matrix, $\mathbf{A}$ is the adjacency matrix, $g(\cdot)$ is the permutation invariant neighborhood aggregation function, $\mathbf{X}_{\mathcal{N}(i)}$ is the multiset of features of the neighborhood $\mathcal{N}(i)$, and $\mathbf{H}$ is the latent feature matrix. Typically the function $g(\cdot)$ includes the node i's own features in the aggregation, here denoted as $\mathbf{x}_i$.

The k-hop neighborhood aggregation is calculated by running the GNN again, but with the latents in place of the original features. It should however be noted, that sufficiently many neighborhood hop aggregations will lead to latent features that are indistinguishable from each other [33, 34]. This is because stacking too many layers of GNN will no longer learn relevant and useful latent features for the nodes. Rather, the features of the nodes become equally similar, or "smoothed", losing all unique expressiveness in the process. This phenomenon is called over-smoothing. Depending if the graph is small, or if the information in the graph is heavily localized, a fewer k-hops might be opted.

3.7.2 Encoding

The permutation invariant function $g(\cdot)$ of the GNN encodes the latent features. This function can be customized. Designing the function $g(\cdot)$ is an intense topic of active research, and multiple interdisciplinary approaches have independently come up with a myriad of different approaches. In order to unify the previous research, Bronstein et al. [9] propose a blueprint to classify the function $g(\cdot)$ into 3 different paradigms, which would cover most of the contemporary approaches. These paradigms are: convolutional, attentional and message-passing GNNs.

Convolutional GNN

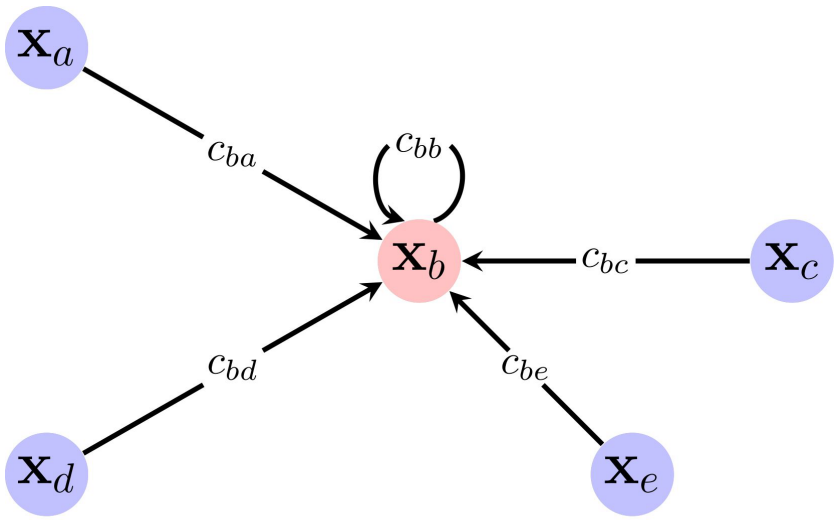


Figure 3.4. *Convolutional GNN [9].*

Convolutional GNN can be viewed as weighted neighborhood aggregation, where the weights are *constant*. Each neighboring node is associated with a constant coefficient, or a weight, which is used to multiply the node's feature vector. The neighborhood aggregation with constants is illustrated in Figure 3.4, where each link in the neighborhood is assigned a single constant, which is used to multiply the features of the corresponding neighbors. The weighted features are then aggregated, making the function of convolutional GNN of form:

$$\mathbf{h}_i^{k+1} = \sigma\left(\bigoplus_{j \in N(i)} c_{ij}\mathbf{h}_j^k\right) \tag{3.7}$$

where the coefficient of node i and j is denoted as c_{ij}.

Typically the weights depend on the adjacency matrix. In the most simplest case, only

the adjacency matrix can be used. For example, if the adjacency matrix is binary, the constant would be 1 for neighbor nodes and 0 to others. The weights can also depend on the adjacency matrix in more complex ways, such as matrix decompositions.

Implementations based on the dependency of the adjacency matrix include: the ChebyNet [35], Graph Convolutional Network (GCN) [36] and Simple Graph Convolution (SGC) [37].

Convolutional GNNs can be advantageous in situations where the graph is homophilious, meaning similar nodes are more likely to be connected with a link [9]. Furthermore, convolutional GNNs are scalable and easy to implement as basic matrix operations, making them especially attractive for large homophilious graphs [9].

Attentional GNN

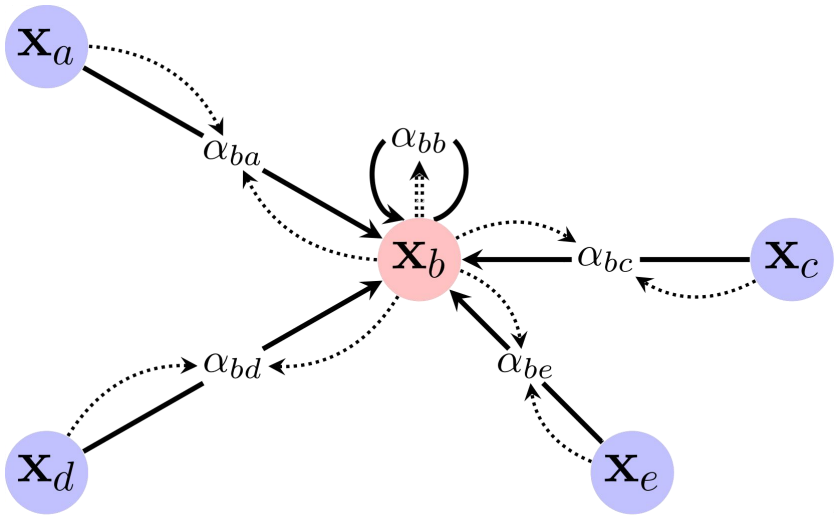


Figure 3.5. *Attentional GNN [9].*

Graphs can encode more complex type of information in their links than homophily [9]. For example, when sharing content on a social media platform, people might share content that they support and agree on but also content they disagree on. Therefore the link does not strictly need to encode similarity, rather it can encode repulsion or more complex relationships.

Attention is a concept that origins from psychological cognitive attention. In humans, cognitive attention emerges as a selection on certain inputs, while ignoring other perceivable information [38]. When applied to neural networks, attention mimics cognitive attention by using coefficients, or weights, as a scale to determine how meaningful the given input is. A greater weight implies greater attention, and therefore a greater importance, while a smaller weight implies a lesser importance.

In the setting of GNN, this can be modeled by replacing the constant weights, as was done in the convolutional GNN, with learnable attentive weights. The function for attentional GNN then becomes:

$$\mathbf{h}_i^{k+1} = \sigma\left(\bigoplus_{j \in N(i)} a(\mathbf{h}_i^k, \mathbf{h}_j^k)\, \mathbf{h}_j^k\right) \tag{3.8}$$

where $a(\mathbf{h}_i^{(k)}, \mathbf{h}_j^{(k)})$ is the attention function that calculates the importance of the features of the neighbor node j to node i. The attentive coefficient for each neighboring node is illustrated in Figure 3.5.

Compared to convolutional GNN, attentional GNN requires more computation and space, since a coefficient needs to be calculated for each link [9]. However, the additional computational cost can be viewed as a reasonable requirement, when it enables to process non-homophilious graphs.

Implementations with attentive weights include: Mixture Model Networks (MoNet) [39], Graph Attention Network (GAT) [3], Gated Attention Network (GaAN) [40].

Message-passing GNN

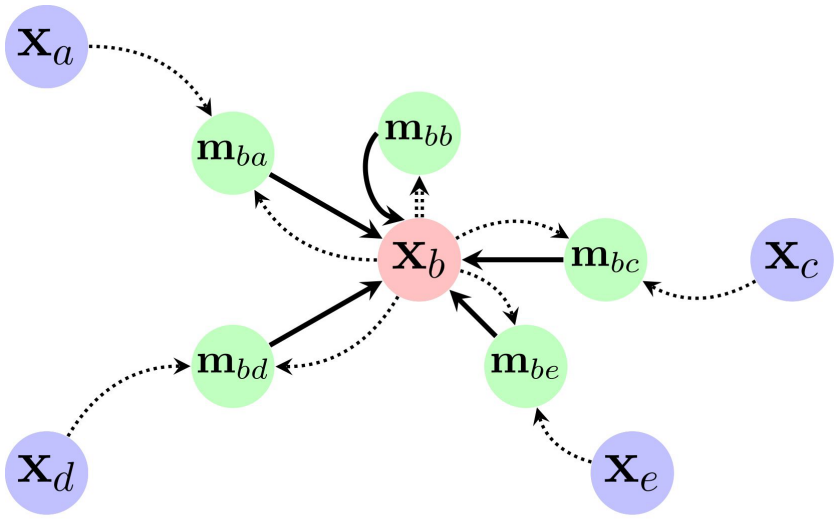


Figure 3.6. *Message-passing GNN [9].*

Sometimes a single weight multiplication on features is not sufficient to capture the complexity encoded in the links. In these cases, the concept of weighing the features is discarded. Instead, an arbitrary vector, or a "message", can be created between the neighboring nodes. The message is then "passed" along with the links to the aggregating node. This process is illustrated in Figure 3.6.

The function takes in the features of the neighboring nodes and outputs an arbitrary message that is then aggregated. The message creating function can be task dependent, making message-passing GNN the most generalized approach. The function of a message-passing GNN is therefore of form:

$$\mathbf{h}_i^{k+1} = \sigma\left(\bigoplus_{j \in N(i)} m(\mathbf{h}_i^k, \mathbf{h}_j^k)\right) \tag{3.9}$$

where $m(\mathbf{h}_i^{(k)}, \mathbf{h}_j^{(k)})$ is the arbitrary function that takes in the features of sender node j and receiver i, and outputs a vector for aggregation.

The benefit of doing this is that a message-passing GNN can model complex phenomenons, like algorithmic reasoning and computational chemistry [9]. The downside is that computing and storing the messages can be expensive and therefore it does not scale well. Learnability issues might also be introduced [20].

Implementations with message-passing GNNs include: Interaction Networks [41], Message Passsing Neural Networks (MPNN) [42] and GraphNets [43].

It should however be noted, that fundamentally both of the previously presented convolutional and attentional GNNs can be reduced to be special cases of message-passing GNNs [9]. In those cases, the arbitrary message-passing function takes the previously discussed form in the convolutional and attentional GNN sections. However, due to their applicability and popularity, Bronstein et al. elevated them as their own categories.

3.7.3 Decoding

Link prediction, classification of nodes, and classification of (sub)graphs are common tasks when dealing with graphs. A decoder can be trained to take the latent features as input and to produce predictions for the task.

In node classification, a decoder can be trained to classify each node:

$$\mathbf{z}_i = f(\mathbf{h}_i) \tag{3.10}$$

where $\mathbf{z}_i$ is the classification prediction for the node, and $\mathbf{h}_i$ is the latent feature vector of node i.

For a graph classification task, the nodes can be aggregated together and given as an

input for the classifier:

$$\mathbf{z}_G = d\left(\bigoplus_{i \in N} \mathbf{h}_i\right) \tag{3.11}$$

where $\mathbf{z}_G$ is the classification prediction for the graph, $\mathbf{h}_i$ is the feature vector of node i, $\bigoplus$ is a typically permutation invariant aggregation function, though an RNN can also be employed [44].

For predicting links between a pair of nodes, the classifier predicts the existence of the link based on the features of both nodes:

$$\mathbf{z}_{ij} = d(\mathbf{h}_i, \mathbf{h}_j) \tag{3.12}$$

where $\mathbf{z}_{ij}$ is the prediction for the link between nodes i and j, $\mathbf{h}_i$ and $\mathbf{h}_j$ are the latent feature vectors of node i and j respectively.

If the graph is undirected, this implies the link is bidirectional. In directed graphs, the feature vectors need to be distinguished in order to predict the direction. One technique is to concatenate the vectors, and predict the existence of the link based on the concatenated vector. The act of concatenation will produce two different vectors based on which order the vectors were concatenated.

3.7.4 Temporal GNNs

Previously mentioned GNN-methods assume the underlying graph is static. In reality, many real-world graphs are fundamentally temporal. It is a much harder problem to create node representations for a dynamic graph compared to a static graph, because the links and nodes can change over time.

To capture the temporal information in the discrete time domain, many have utilized recurrent neural networks (RNNs) along with the GNN. In this setting, the GNN generates the node feature matrix for each graph snapshot that are then fed into an RNN. This setup allows the RNN to process the node embeddings in sequence and to mine the temporal information between the snapshots. This is also called spatio-temporal modeling. Different variations of GNN and RNN can be used. The final output of the RNN is a graph embedding that has learned both the structural and temporal information of the graph. This graph embedding can then be used in downstream tasks. A generalization of the GNN+RNN setup is illustrated in Figure 3.7.

Implementations with the GNN+RNN setup include: GCRN [45], STGCN [46], GC-LSTM [47], T-GCN [48], and TGC-LSTM [49].

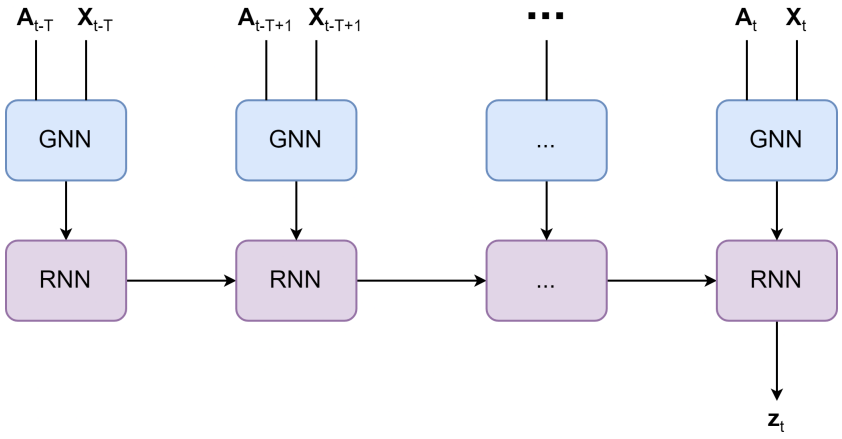


Figure 3.7. *A generalization of the GNN+RNN setup. The structural information of the graph is captured with a GNN. The output of the GNN is the input to an RNN, which in turn captures the temporal information in the graph.*

Pareja et al. [50] point out that in order for GNN+RNN to function properly, the node set must be known now, and in the future. This is not always a realistic assumption in real-world networks. Repeated changes in the node set make the GNN+RNN setup less desirable. To tackle this problem, Pareja et al. propose EvolveGCN [50], which modifies the weights of the GNN with an RNN instead of modifying the node embeddings with an RNN.

Overall, the research in deep learning and GNNs on temporal graphs is still new and developing. This is especially true with continuous temporal graphs, which is out of scope of this thesis. The field is advancing quickly, and currently best performing results in discrete time domain have been produced with GNNs and RNNs. They are next employed in a link prediction task with the temporal network dataset of this thesis.

4. Experiments

4.1 Dataset

The dataset for this thesis is a binary money flow network gathered from the Helsinki Stock Exchange, between the time span of 2000-01-31 to 2008-12-21. The nodes in the network are stocks that were publicly listed and active for trading during the complete time span. The names of the stocks are listed in Appendix A.

4.1.1 Statistics

The summary statistics of the snapshots of the money flow network can be seen in Table 4.1. The statistics are obtained by first calculating the respective statistic for an individual snapshot, or in the graph connectivity case, from 2 consecutive snapshots. The summary statistics are then aggregated by the said aggregation function in the table.

The link probability is calculated by:

$$\text{Link probability} = \frac{|L_t|}{N^2} \tag{4.1}$$

where $|L_t|$ is the number of links at time t, and N is is the number of stocks in the network.

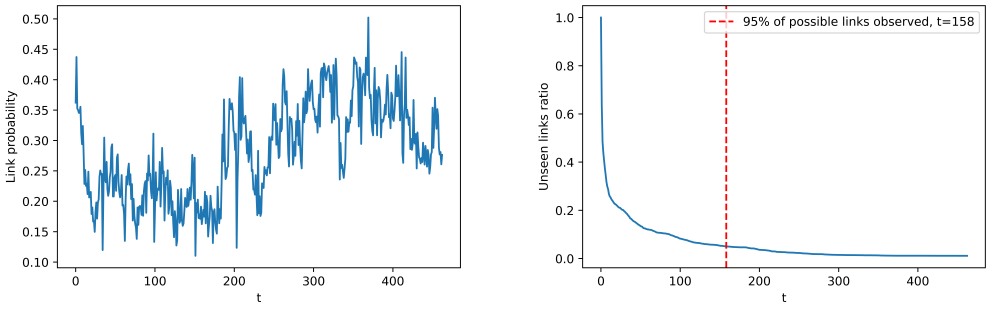
The connectivity function calculates, how many links have remained unchanged between two consecutive snapshots, and divides that number with the number of all possible links. A link remains unchanged, if the value of the element in the adjacency matrix is the same in both of the consecutive snapshots. The connectivity function between two consecutive graph snapshots is defined as:

$$\text{Connectivity}(G_t, G_{t+1}) = \frac{\mathbf{e}^\top \mathbf{B} \mathbf{e}}{N^2} \tag{4.2}$$

where $\mathbf{B}$ is the binary elementwise equality comparison matrix between the adjacency matrices of the two consecutive snapshots, $\mathbf{e}$ is a vector of ones, and N is the number of stocks in the network.

Time steps (weeks)	463
N (nodes)	93
Degree mean	26.2
Links mean	2400
Links std	690
Link probability mean	0.28
Connectivity(G_t, G_{t+1}) mean	0.78
Connectivity(G_t, G_{t+1}) std	0.04

Table 4.1. *Summary statistics of the snapshots of the money flow network.*



(a) Link probability at time t.

(b) Percentage of unseen links at time t.

Figure 4.1. *Links over time in the money flow network.*

Since the money flows are created from any amount of money flow, it could be expected that it would ultimately lead to a fully, or nearly fully connected network, where every pair of stock would be linked bidirectionally. However, this is not the case with the Helsinki Stock Exchange during this time period. The maximum amount of possible links in this network (self-loops allowed) is: $N * N = 93 * 93 = 8649$. The amount of links observed is not even close to the maximum, as is illustrated with link probability at any given time step t in Figure 4.1(a). Some other stock exchanges could have a different situation, and it would then be necessary to somehow design more advanced conditions for the existence of money flow. For example, to set thresholds for the number of investors required, amount of money transferred, or by using some statistical methods to validate the links against a null hypothesis.

The connectivity between two consecutive graph snapshots is on average 0.78 similar, with standard deviation of 0.04. This means that there is a relatively strong base predictability present in the previous snapshot for predicting the connectivity of the next snapshot.

Uncharacteristically to many temporal networks, the money flow network does not possess a high class imbalance between the previously observed links and the never observed links. Overwhelmingly many links have been observed at least once, as can be seen in

Feature	Description
Revenue	The sum of daily natural logarithmic revenues over a week. Daily natural logarithmic revenue $\log(p_t/p_p)$ is calculated by taking the natural logarithm of the quotient of today's last paid price p_t divided by the previous market open day's last paid price p_p.
Market open	Number of days the market has been open during the week.
Shares outstanding	Weighted mean of the company's shares outstanding. For example, the market is open for 5 days, and for 4 days the number of shares is 100, and for 1 day 200: $(4 * 100 + 1 * 200)/5 = 120$.
Volatility	Standard deviation of daily natural logarithmic revenues during the week.
Volume	Number of stocks traded during the week.
Trade count	Number of trades during the week.

Table 4.2. *Feature descriptions.*

Figure 4.1(b). 95 percent of all possible links have been observed at least once by the time step 158. At the end of the dataset (t = 463), 99 percent of the links have been observed at least once. The absence of class imbalance in this dataset is most likely due to the loose criterion of what constitutes a money flow.

4.1.2 Input Features

Along with the weekly money flows, weekly features are observed. There are 6 temporal features: revenue, number of open market days, shares outstanding, volatility, volume, and the number of trades. Their meanings are described in Table 4.2. The natural logarithm of revenue is used to make the revenue additive over time. The number of shares outstanding typically does not fluctuate as often as the other features. However, in case of the number of shares outstanding changing during the week, a weighted mean is calculated.

The correlation matrix of the averaged input features using Pearson correlation can be seen in Table 4.3. Note, that the in- and out degrees of the correlation matrix are not considered as input features and are not used to train the model. The input features are averaged over all the snapshots in the dataset. It can be seen that volume, trade count, and shares outstanding are strongly positively correlated with each other. Revenue and volatility are negatively correlated. In- and out-degree are strongly positively correlated. Both in- and out-degree are somewhat correlated with shares outstanding, volume, and trade count. Rest of the features and degrees are substantially less correlated. Pearson's correlation coefficient is defined as:

$$P_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\mathrm{std}(X)\mathrm{std}(Y)} \tag{4.3}$$

where $P_{X,Y}$ is the Pearson correlation coefficient and $\mathrm{cov}(X,Y)$ is the covariance be-

	Revenue	Market open	Shares	Volatility	Volume	Trade count	In-degree	Out-degree
Revenue	1	0.092	-0.037	-0.675	-0.041	-0.014	-0.008	0.001
Market open	0.092	1	0.027	-0.032	0.024	0.033	0.137	0.132
Shares	-0.037	0.027	1	0.063	0.991	0.975	0.4	0.399
Volatility	-0.675	-0.032	0.063	1	0.082	0.011	-0.076	-0.092
Volume	-0.041	0.024	0.991	0.082	1	0.978	0.385	0.382
Trade count	-0.014	0.033	0.975	0.011	0.978	1	0.501	0.5
In-degree	-0.008	0.137	0.4	-0.076	0.385	0.501	1	0.998
Out-degree	0.001	0.132	0.399	-0.092	0.382	0.5	0.998	1

Table 4.3. *Correlation matrix of the averaged features and node degrees. The node degrees are not used in the training of the model.*

tween the features X and Y respectively, $\mathrm{std}(\cdot)$ is the standard deviation.

Normalization (scaling) of the input features is a common practice with neural networks because it helps to stabilize the learning process [51]. In this dataset, the features are normalized with min-max normalization. The complete time span of 463 weeks is considered when determining the maximum and minimum value of a feature. Min-max normalization is defined as:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{4.4}$$

where x is a feature and x' is the normalized feature.

Based on all of the previous, it can be summarized that the network is a fully observed temporal binary network, which is observed as snapshots at discrete time intervals. The number of nodes is fixed, the links are temporal, and the node features are temporal. The money flow network can be denoted as $G = \{V, E_t, \mathbf{X}_t\}$, where V is the set of static nodes, E_t is the set of links and $\mathbf{X}_t$ is the node feature matrix at time t respectively.

4.2 Methodology

Since the network dataset is observed discretely, only models that require discrete observations of the network are considered. Continuous models may be employed in discrete settings, but for all practical purposes, they can be ignored. Another thing that can be ignored is the concern about repeated changes in the node set and its effect on the quality of the node representations. This is because the node set is static.

The framework for this link prediction task is the GNN+RNN setup. The main influence for this methodology stems from the works of Zhao et al. (T-GCN) [48], where they focused on predicting traffic volumes in a traffic network, and Li et al. (TSAM) [52], where they focused on link prediction on directed graphs with autoencoders, a temporal

attention mechanism, and adjacency matrix transformations.

The encoder of T-GCN encodes the temporal network into one single latent vector, as is typical for a GNN+RNN setup. As for the decoder, T-GCN uses a fully connected network, which is trained along the model. The authors used T-GCN to predict traffic volumes between nodes. With a proper choice of the loss function, this concept can be converted to predict connections (links) between nodes. This would end up being the autoencoder architecture, where the input and output are of similar shape.

Li et al. (TSAM) used the autoencoder architecture to decode the latent graph feature vector into an asymmetric matrix, which represents a directed graph. The autoencoder architecture solves the problem of predicting the direction of the link, which is an extra challenge with directed networks. Mainly because of this reason, the autoencoder is chosen as the architecture for the model in this thesis.

Out of the possible GNN-paradigms (convolutional, attentional, message-passing GNNs), the convolutional GNNs are not well suited for this task as the GNN. There are two reasons for this. Firstly, many successful convolutional methods are only available for undirected networks. Therefore, they would not be suited for the money flow network, which is directed. Secondly, the money flow network cannot easily be justified to be homophilious, and for the links to encode similarity. On the contrary, the stocks might be viewed to possess a repulsion to each other. For example, it could be viewed that a certain stock is no longer favorable to own, and is thus sold, in order to buy a more favorable stock. This interaction would not encode similarity between the stocks, but rather a dissatisfaction to the stock sold, which manifests itself as a money flow to another stock.

From the 2 remaining GNN paradigms, the attentional GNN is chosen for this work over the message-passing GNN. This is because in order to answer the research questions posed, the simpler attentional GNN is sufficient to reach conclusions. Moreover, it is not entirely clear, how the message-passing function should be designed for this dataset. It is also not clear, what benefits it would possess, compared to the single weighted attentional approach. Therefore, developing a more complex message-passing function to model the relationships in the dataset is left for future research.

The chosen variant for the attentional GNN is the Graph Attention Network (GAT) [3]. For the RNN, the Gated Recurrent Unit (GRU) [4] is chosen as the variant.

In order to justify the significance of temporal modeling as part of the model, a simpler, competing ablation model is additionally trained. The encoder of the ablation model consists only of a static GAT-layer to model the structural dependency without modeling the temporal dependency. The overview of the GAT+GRU model is depicted in Figure 4.2, and the static-GAT ablation model in Figure 4.3.

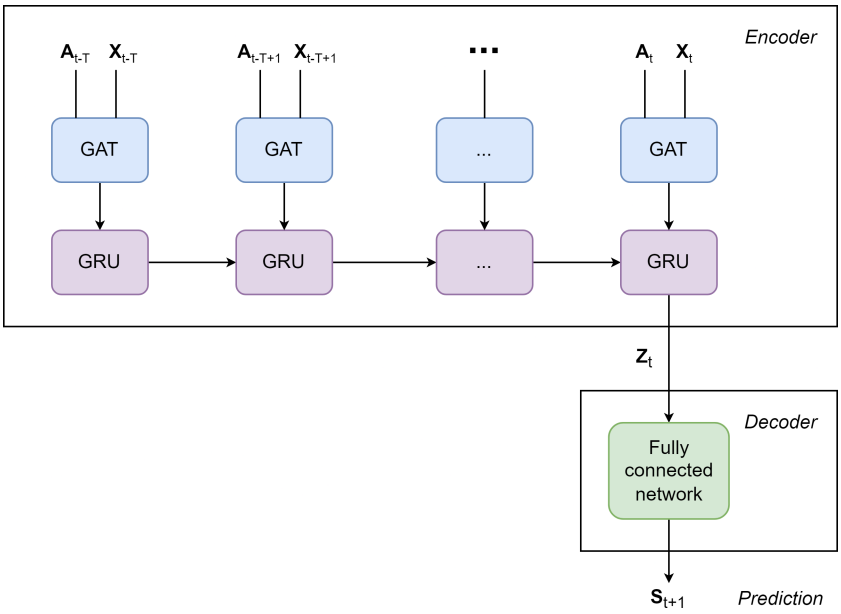


Figure 4.2. *Overall architecture of the GAT+GRU model.*

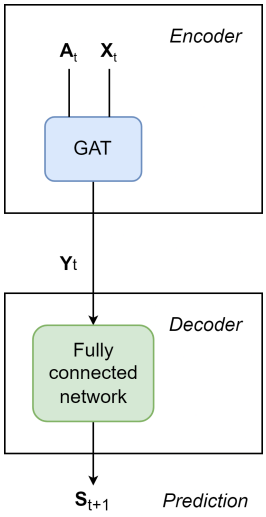


Figure 4.3. *Overall architecture of the static-GAT ablation model.*

Since the solution is an autoencoder architecture, the size of the latent dimension can have impact on the performance of the model. Therefore, its impact is tested with multiple values.

To assess the importance of the input features, both models are run with and without the input features. To test the model without input features, a vector of ones is fed to the model in place of the original input features. The size of this vector is 6, which corresponds to the number of original input features. This setup is analogous to smoothed features, where no information is encoded in the input features.

4.2.1 Structural Modeling (GAT)

GAT [3] takes the adjacency matrix $\mathbf{A}_t$ and the input feature matrix $\mathbf{X}_t$ as input. GAT performs attentional aggregation of the node neighborhoods by first applying a linear transformation to every node. After that, a shared attention mechanism a calculates attention weights between pairs of nodes to distinguish the importance of the features of node j to node i. GAT is agnostic to the choice of the attention mechanism. The authors of GAT used a single layer feedforward neural network as the attention mechanism, and this approach is opted here. The complete attention weight e_{ij} is calculated as:

$$e_{ij} = \mathrm{LeakyReLU}\left(\mathbf{a}^T \cdot \mathrm{Concat}(\mathbf{W}\mathbf{x}_i, \mathbf{W}\mathbf{x}_j)\right) \tag{4.5}$$

where $\mathbf{W}$ is the weight matrix for the linear transformation, $Concat(\cdot, \cdot)$ is the vector concatenation operation, $\mathbf{a}^T$ is the transposed weight vector for the single layer feedforward neural network attention mechanism, and $\mathrm{LeakyReLU}(\cdot)$ is a non-linear activation function with negative input slope of 0.2.

Not every pair of nodes gets an attention weight however. Ignoring some pairs on a certain criterion is called masked attention. In this thesis, the attention weight is calculated between node i in its 1-hop *in*-neighborhood. The in-neighborhood of node i consists of the neighboring nodes of node i, that are connected to node i with an incoming link. The neighborhood needs to be defined because the dataset network is directed, which causes the in- and out-neighborhoods to be different. More formally, the attention weight is calculated for nodes $j \in \mathcal{N}_{in}(i)$, where $\mathcal{N}_{in}(i)$ is the 1-hop in-neighborhood of node i.

To make the attention weights comparable, they are normalized with the softmax function, denoted as:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_{in}(i)} \exp(e_{ik})} \tag{4.6}$$

where α_{ij} is the normalized attention weight.

The output of GAT is a new latent feature matrix $\mathbf{Y}_t = \{\mathbf{y}_1^t, \mathbf{y}_2^t, \ldots, \mathbf{y}_N^t\}$, where t is the time step. The size of the latent dimension is F^*. In this work, the matrix $\mathbf{Y}_t$ is flattened in both GAT+GRU and static-GAT models in order to feed it forward in the model pipeline.

4.2.2 Temporal Modeling (GRU)

The GRU consists of a sequence of GRU cells that take two inputs: the main input and the output of the previous GRU cell [4]. Here the main input is the flattened latent feature matrix from GAT. The output of a GRU cell at time t is a vector $\mathbf{z}_t$. The size of the vector in this work is predetermined to be the same size as the latent node features F^* in GAT . When $t = 0$, there is no previous output from a GRU cell, and in its place a vector of zeros is used. The output vector $\mathbf{z}_t$ of the GRU cell can be denoted as:

$$\mathbf{a}_t = \sigma(\mathbf{W}_a \mathbf{y}_t + \mathbf{U}_a \mathbf{z}_{(t-1)} + \mathbf{b}_a) \tag{4.7}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r \mathbf{y}_t + \mathbf{U}_r \mathbf{z}_{(t-1)} + \mathbf{b}_r) \tag{4.8}$$

$$\mathbf{n}_t = \tanh(\mathbf{W}_n \mathbf{y}_t + \mathbf{a}_t \odot \mathbf{W}_n \mathbf{z}_{(t-1)} + \mathbf{b}_n) \tag{4.9}$$

$$\mathbf{z}_t = (1 - \mathbf{r}_t) \odot \mathbf{n}_t + \mathbf{r}_t \odot \mathbf{z}_{(t-1)} \tag{4.10}$$

where $\mathbf{W}, \mathbf{U}, \mathbf{b}$ are learnable parameters of the respective gates for update gate $\mathbf{a}$, reset gate $\mathbf{r}$, and the new memory gate $\mathbf{n}$ at time t, $\sigma(\cdot)$ is the sigmoid function, $\odot$ represents the Hadamard product.

4.2.3 The Decoder Network

The output of the encoder is fed into the decoder. In GAT+GRU, this is the output of the last GRU cell $\mathbf{z}_t$. In static-GAT, the output of the encoder is the flattened latent feature matrix $\mathbf{Y}_t$. The decoder network consist of 2 layers, and is denoted as:

$$d(\mathbf{h^*}_t) = \text{sigmoid}\left(\text{LeakyReLU}(\mathbf{h^*}_t \mathbf{W}_{h^*} + \mathbf{b}_{h^*})\mathbf{W}_o + \mathbf{b}_o\right) \tag{4.11}$$

where $\mathbf{h^*}_t$ is the output of the encoder (either GAT+GRU or static-GAT), $\mathbf{W}_{h^*}$ and $\mathbf{W}_o$ are the weights of the latent layer and the final output layer of the decoder network respectively.

The output of the decoder network is another vector, which is reshaped into a matrix of shape $N \times N$, where N is the number of nodes in the network. The final output of the autoencoder is a score matrix $\mathbf{S}_{t+1}$, which represents the prediction of links existing in the next graph snapshot G_{t+1}. The elements inside the matrix are between 0 and 1 and represent the model's confidence for the link to exist.

The decoder network is similar in TSAM. However, in TSAM, both activation functions in the decoder are ReLUs. In this work, the outer activation function is sigmoid, and inner is LeakyReLU. During the experiments it became apparent, that these activation functions

yielded more stable and faster results compared to ReLU. Sigmoid squishes the output between [0,1] and is therefore directly usable as the final score for the corresponding links.

4.2.4 Optimization

Li et al. stated in TSAM, that the score matrix $\mathbf{S}_{t+1}$ should be geometrically close to the ground truth adjacency matrix $\mathbf{A}_{t+1}$. This means, that if the link exists in $\mathbf{A}_{t+1}$, the corresponding score in $\mathbf{S}_{t+1}$ should approach 1 for a good prediction and 0 otherwise. In TSAM, they used Frobenius norm to calculate the distance between the two matrices, and this approach is is also used in this work. Frobenius norm of a matrix is the square root of the sum of the squared elements.

Regularization is utilized to penalize the model for complexity [53]. Excess complexity can cause the model to learn the training data too well without the ability to generalize to unseen data. This phenomenon is called overfitting, and regularization aids to prevent it. The model is optimized by the following loss function:

$$\text{loss} = \|(\mathbf{S}_{t+1} - \mathbf{A}_{t+1})\|_F + \lambda\|\theta\|_2^2 \tag{4.12}$$

where θ contains all the trainable parameters of the model, and $\lambda\|\theta\|_2^2$ is the $\mathcal{L}_2$ regularizer.

$\mathcal{L}_2$ regularizer takes the norm of the squared weights of the model and adds it to the loss. $\mathcal{L}_2$ regularizer aims penalize large weights, since they are indicative of overfitting [54]. λ, sometimes called *weight decay*, is a predetermined hyperparameter for the $\mathcal{L}_2$ regularizer, which scales the importance of the regularizer in the loss function. The loss function is optimized with the Adaptive Moment Estimation (Adam) optimizer [55].

4.2.5 Evaluation Metrics

The model is evaluated with the best performing parameters, which were detected during the validation. Since the money flow network dataset does not suffer from a severe class imbalance, the performance of the model can be evaluated with a single evaluation metric. The chosen metric is therefore the area under the receiver operating characteristic curve (AUROC). This choice implies, that every link and non-link is considered equally worthy of predicting and is given equal weight in the evaluation.

4.2.6 Baseline

The model should be able to perform better than a naive baseline prediction. As for the baseline, the current graph observation acts as the prediction for the next snapshot. The baseline is denoted as $H_0 : G_t = G_{t+1}$.

This baseline is chosen for two reasons. Firstly, even though the baseline is by definition unable to predict new (unseen) links, it is not vital in this dataset, because the typical class imbalance between previous and new links is not present. This makes it a viable baseline in the first place. Secondly, the connectivity between two consecutive snapshots remains almost 80% similar with only little deviation. This can be argued to be a relatively good base predictability.

4.3 Setup

4.3.1 Programming

The chosen programming language for this work is Python [56]. Python is chosen because Python encompasses a well-rounded and well-established collection of data science and machine learning libraries.

The back-bone libraries for the work are PyTorch [57, 58] and PyTorch Geometric (PyG) [59, 60]. PyTorch is an open source machine learning and deep learning framework, which provides an object oriented approach for programmers to implement machine learning in Python. PyTorch Geometric is a geometric deep learning framework, which focuses mainly on deep learning methods for graphs, such as GNNs and their implementations. For the implementation of graph attention network (GAT), the implementation from PyG is used. For the GRU, the implementation from PyTorch is used.

4.3.2 Dataset Partitioning

The dataset is partitioned into training, validation, and testing sets. The training set contains 313 snapshots, validation set 50 snapshots, and testing set 100 snapshots. Together they sum up to the length of the dataset, 463 snapshots. The chronological order of the graph snapshots persists between the partitions. This means that the snapshots in the training set are the earliest, and snapshots in the testing set are the newest. Respectively, the chronological order persists within a given partition.

The length of a partition is denoted as T. For the GAT+GRU -model, the model is given a sliding window of data from the start of the partition up to the snapshot t. The static-GAT -model is only given a single snapshot t. Both models predict the snapshot $t + 1$ for each $t = 1, ..., T$.

Training set is used to train the model. Training the model over the training set once is called an epoch. Training the model over multiple epochs improves the performance of the model. However, training the model for too many epochs will result in overfitting and bad generalization for unseen data.

Finding the optimal number of epochs for training is a difficult problem. To alleviate this, the performance of the model is validated after each epoch with the validation set, which was not used in training. Validation enables to keep track of the model parameters (weights) of the best performing epoch, and to execute early stopping in order to avoid overfitting [61].

The principle of early stopping is to stop the training when the validation loss no longer decreases. The model is given a chance to decrease the loss for a fixed number of consecutive epochs, called patience. If the loss decreases during the patience period, the patience resets back to its original value. If not, the training is stopped and the model parameters from the epoch with the lowest validation loss are selected as the final parameters of the model. Lastly, the testing set is used to evaluate the model with the final parameters.

4.3.3 Hyperparameters

Hyperparameter is a parameter in the model, that typically cannot be estimated or learned from the data itself. Instead, tuning the hyperparameter is left for the experimenter's discretion. Usually, tuning the hyperparameters requires running the model multiple times with different values for the hyperparameters.

The hyperparameters for this work are the following.

Hyperparameter	Values
Latent space size	[2, 4, 8, 16, 32, 64, 128, 256]
λ	$5 * 10^{-7}$
Learning rate	GAT+GRU: 0.01 | static-GAT: 0.001
Patience	40

Table 4.4. *Hyperparameters.*

5. Results

All of the experiments on the models are run independently 5 times. The AUROC and loss of the models are reported as their average of those 5 experiments along with standard deviation. AUROC is reported in Table 5.1 and loss in Table 5.2. Both AUROC and loss are reported based on the dimension of the latent space. The results for AUROC are also plotted in Figure 5.1(a), and for loss in Figure 5.1(b).

Overall, it can be seen the size of the latent feature space matters in the results. However, even the lowest value (2) for the latent features is able to reach over 81 AUROC in all of the models. GAT+GRU -model performs better with fewer latent features than static-GAT. The highest AUROC score of 83.6 $\pm$ 0.1 is reached with GAT+GRU without features when the latent dimension size is 8, and with features when the dimension size is 64. However, the margins are extremely slim between all of the models as the size of the latent feature space is 128.

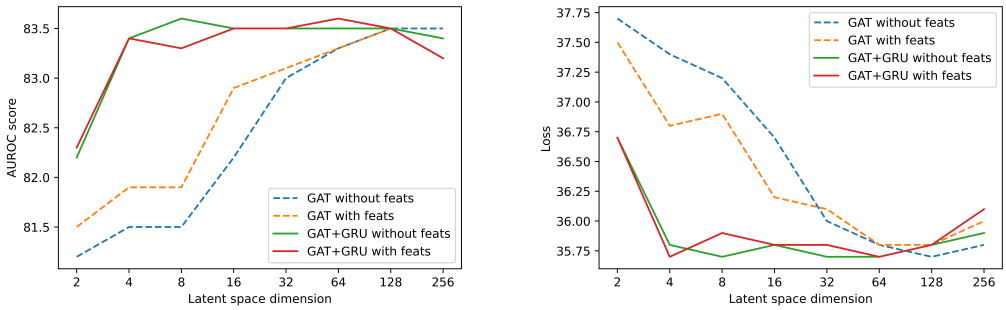
It is not outruled that even higher dimensions of the latent space could reach better results. However, this is unlikely. The computational complexity increases drastically, when the size keeps increasing. Too much complexity generally leads to poorer performance after some point. In this work, the computer that was used to conduct this research could not process dimensions greater than 256. If higher dimensions are necessary to use, a possible remedy can be to to use LayerNorm [62] to normalize weights between layers. This is because normalizing the weights between layers can have the impact of reducing the computational burden. In this work, LayerNorm was not used.

Dim	GAT w/o feats	GAT with feats	GAT+GRU w/o feats	GAT+GRU with feats
2	<u>81.2</u> $\pm$ 0.1	<u>81.5</u> $\pm$ 0.6	<u>82.2</u> $\pm$ 0.9	<u>82.3</u> $\pm$ 0.9
4	81.5 $\pm$ 0.3	81.9 $\pm$ 1.0	83.4 $\pm$ 0.1	83.4 $\pm$ 0.1
8	81.5 $\pm$ 0.3	81.9 $\pm$ 0.9	**83.6** $\pm$ 0.1	83.3 $\pm$ 0.2
16	82.2 $\pm$ 0.1	82.9 $\pm$ 0.3	83.5 $\pm$ 0.1	83.5 $\pm$ 0.1
32	83.0 $\pm$ 0.0	83.1 $\pm$ 0.1	83.5 $\pm$ 0.0	83.5 $\pm$ 0.0
64	83.3 $\pm$ 0.0	83.3 $\pm$ 0.1	83.5 $\pm$ 0.1	**83.6** $\pm$ 0.1
128	**83.5** $\pm$ 0.0	**83.5** $\pm$ 0.0	83.5 $\pm$ 0.1	83.5 $\pm$ 0.1
256	**83.5** $\pm$ 0.1	83.2 $\pm$ 0.2	83.4 $\pm$ 0.2	83.2 $\pm$ 0.3

Table 5.1. *AUROC of the models with different sizes of latent feature dimension. Highest (best) values are in bold, lowest are underscored.*

Dim	GAT w/o feats	GAT with feats	GAT+GRU w/o feats	GAT+GRU with feats
2	37.7 ± 0.3	37.5 ± 0.7	36.7 ± 0.8	36.7 ± 0.8
4	37.4 ± 0.4	36.8 ± 0.7	35.8 ± 0.1	**35.7** ± 0.1
8	37.2 ± 0.2	36.9 ± 0.6	**35.7** ± 0.1	35.9 ± 0.1
16	36.7 ± 0.1	36.2 ± 0.2	35.8 ± 0.1	35.8 ± 0.0
32	36.0 ± 0.0	36.1 ± 0.1	**35.7** ± 0.0	35.8 ± 0.1
64	35.8 ± 0.0	**35.8** ± 0.0	**35.7** ± 0.1	**35.7** ± 0.1
128	**35.7** ± 0.0	**35.8** ± 0.1	35.8 ± 0.1	35.8 ± 0.1
256	35.8 ± 0.0	36.0 ± 0.1	35.9 ± 0.1	36.1 ± 0.1

Table 5.2. *Loss of the models with different sizes of latent feature dimension. Lowest (best) values are in bold, highest are underscored.*



(a) AUROC (b) Loss

Figure 5.1. *AUROC (a) and loss (b) of the models with different latent space dimension sizes.*

RQ1: Baseline. The AUROC performance of the baseline $G_t = G_{t+1}$ is 73.0. Both Static-GAT and GAT+GRU outperform the baseline by significant margins. The models are able to learn more complex and meaningful relationships in the data, compared to the baseline of predicting the next snapshot equaling the previous snapshot.

RQ2: Input features. Both of the models yield similar results with or without the input features. This is somewhat surprising. It can be concluded that the available input features do not improve the performance of the models.

The available input features are general information about how the securities are traded in the market. Because of this, one possible reason for this result is that the connectivity of a snapshot already contains the information encoded in the general features of the securities. Thus, making the input features redundant.

RQ3: Modeling temporal evolution. Both Static-GAT and GAT+GRU -models yield similar results. It can therefore be concluded that modeling the temporal evolution does not improve the performance of the model. This is not as surprising as it seems. Compared to traffic volume predictions, a stock market does not have clear temporal seasonalities, like yearly seasons and day-night cycles. On the contrary, stock markets are widely known for their independence of timing.

6. Suggestions For Future Research

The GNN+RNN setup allows different variations. To name a few possibilities: GRU can be replaced with LSTM, more GNN-layers can be added, the attentional GNN can be replaced with a custom message-passing function, and an attention mechanism can be applied on temporal level. However, as demonstrated in this thesis, modeling the temporal evolution should be ablated from the model, and compared to a model with structural modeling only. This is because the extra complexity is not guaranteed to mine information in the temporal evolution. This would also adhere to the principles of machine learning, where in the event of two models with different complexities perform similarly, the model with less complexity is favored.

Tweaking the GNN+RNN setup might not be the realistic path for future research for this dataset. For example, it is hard to imagine replacing GRU with LSTM would yield significantly better results than the static-GAT. The difference between the temporal and static modeling in this thesis should have been clearly visible in order to warrant the extra adjusting of the temporal model.

Instead of tweaking the GNN+RNN setup, the dataset itself could be modified. The criterion for money flow is loose in this dataset and could be tightened. The links could be created more meaningful by applying thresholds on the number investors involved, or the amount of money transferred, or by validating the links to be statistically significant. In these scenarios, it might be possible that modeling temporal evolution of the graph, and the presence of the input features would be of aid to the performance of the model.

Another scenario that might arise from more strictly defined money flows, is that a bigger portion of the links would remain unseen. This would open up the challenge of predicting the appearance of previously unseen links, which is characteristic to many real-world temporal networks, and ultimately to yield more relevant insight of the money flows in the Helsinki Stock Exchange.

7. Conclusion

In this thesis, Graph Neural Networks (GNNs) were used to study the temporal link prediction problem in a binary money flow network between stocks in the Helsinki Stock Exchange. The goal was to establish the base predictability in the money flow network using a suitable GNN-based method. A money flow from one stock to another indicated, if there has been any flow of money during the observation period. The money flow network was gathered between years 2000 and 2008, and was observed in weekly snapshots. Additionally, weekly features for the stocks were observed, and used as an input to the model.

Based on the analysis made for the suitable GNN-paradigms and methods, the chosen model consisted of a combination of the Graph Attention Network (GAT) to model the structural dependency, and the Gated Recurrent Unit (GRU) to model the temporal dependency. In order to distinguish the importance of the temporal modeling as part of the model, an ablation model without the temporal modeling was additionally trained. Autoencoder architecture was used to solve the problem of predicting the direction of the links. A baseline model was used to contrast the performance of the used model. The baseline predicted the previous snapshot to be equal to the next snapshot: $G_t = G_{t+1}$.

The performance of the model was evaluated with the area under the receiver operating characteristics curve (AUROC). This is because the money flow network did not exhibit the typical class imbalance between previously seen and never seen links, and therefore all possible links were evaluated to be equally worthy of predicting.

The used model outperformed the naive baseline by clear margins. However, the model did not benefit from the input features, and performed equally well without them. Moreover, the model did not benefit from modeling the temporal dependency, and performed equally well by modeling the structural dependency only.

REFERENCES

[1] Karaila, J. Money Flows Between Securities: Network Analysis in a Stock Market. Tampere University, 2021.

[2] Michelucci, U. An Introduction to Autoencoders. (Jan. 2022). URL: https://arxiv.org/abs/2201.03898v1.

[3] Veličković, P., Casanova, A., Liò, P., Cucurull, G., Romero, A. and Bengio, Y. Graph Attention Networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings* (Oct. 2017). URL: https://arxiv.org/abs/1710.10903v3.

[4] Cho, K., Merriënboer, B. V., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. and Bengio, Y. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference* (June 2014), pp. 1724–1734. DOI: 10.3115/V1/D14-1179. URL: https://arxiv.org/abs/1406.1078v3.

[5] Borgatti, S., Everett, M. and Johnson, J. *Analyzing Social Networks*. 2nd ed. SAGE, 2018, p. 20.

[6] Newman, M. *Networks*. 2nd ed. Oxford University Press, 2018, p. 100.

[7] Holme, P. and Saramäki, J. Temporal Networks. *Physics Reports* 519 (3 Aug. 2011), pp. 97–125. DOI: 10.1016/j.physrep.2012.03.001. URL: http://arxiv.org/abs/1108.1780.

[8] Rossi, E., Chamberlain, B., Fabrizio, T., Twitter, F., Twitter, D. E., Monti, F. and Twitter, M. B. Temporal Graph Networks for Deep Learning on Dynamic Graphs. (June 2020). URL: https://arxiv.org/abs/2006.10637v3.

[9] Bronstein, M. M., Bruna, J., Cohen, T. and Veličković, P. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. (Apr. 2021). URL: http://arxiv.org/abs/2104.13478.

[10] Junuthula, R. R., Xu, K. S. and Devabhaktuni, V. K. Evaluating Link Prediction Accuracy on Dynamic Networks with Added and Removed Edges. *Proceedings - 2016 IEEE International Conferences on Big Data and Cloud Computing, BDCloud 2016, Social Computing and Networking, SocialCom 2016 and Sustainable Computing and Communications, SustainCom 2016* (July 2016), pp. 377–384. DOI: 10.1109/BDCloud-SocialCom-SustainCom.2016.63. URL: https://arxiv.org/abs/1607.07330v1.

[11] Yang, Y., Lichtenwalter, R. N. and Chawla, N. V. Evaluating Link Prediction Methods. *Knowledge and Information Systems* 45 (3 May 2015), pp. 751–782. ISSN: 02193116. DOI: 10.1007/s10115-014-0789-0. URL: https://arxiv.org/abs/1505.04094v1.

[12] Marsland, S. *Machine learning: An algorithmic perspective*. 2014. DOI: 10.1201/b17476.

[13] Fauske, K. M. *https://texample.net/tikz/examples/neural-network*. 2006.

[14] Elman, J. L. Finding structure in time. *Cognitive Science* 14 (2 Apr. 1990), pp. 179–211. ISSN: 0364-0213. DOI: 10.1016/0364-0213(90)90002-E.

[15] Bishop, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, 2006. ISBN: 0387310738.

[16] Bengio, Y., Courville, A. and Vincent, P. Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8 June 2012), pp. 1798–1828. ISSN: 01628828. DOI: 10.1109/TPAMI.2013.50. URL: https://arxiv.org/abs/1206.5538v3.

[17] Lecun, Y., Bengio, Y. and Hinton, G. Deep learning. *Nature 2015 521:7553* 521 (7553 May 2015), pp. 436–444. ISSN: 1476-4687. DOI: 10.1038/nature14539. URL: https://www.nature.com/articles/nature14539.

[18] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *undefined* 86 (11 1998), pp. 2278–2323. ISSN: 00189219. DOI: 10.1109/5.726791.

[19] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C. and Sun, M. Graph Neural Networks: A Review of Methods and Applications. *AI Open* 1 (Dec. 2018), pp. 57–81. ISSN: 26666510. DOI: 10.1016/j.aiopen.2021.01.001. URL: https://arxiv.org/abs/1812.08434v6.

[20] Bronstein, M. M., Bruna, J., Lecun, Y., Szlam, A. and Vandergheynst, P. Geometric deep learning: going beyond Euclidean data. *IEEE Signal Processing Magazine* 34 (4 Nov. 2016), pp. 18–42. ISSN: 10535888. DOI: 10.1109/msp.2017.2693418. URL: https://arxiv.org/abs/1611.08097v2.

[21] Huang, L., Wang, W., Chen, J. and Wei, X. Y. Attention on Attention for Image Captioning. *Proceedings of the IEEE International Conference on Computer Vision* 2019-October (Aug. 2019), pp. 4633–4642. ISSN: 15505499. DOI: 10.1109/ICCV.2019.00473. URL: https://arxiv.org/abs/1908.06954v2.

[22] Rumelhart, D. E. and McClelland, J. L. Learning Internal Representations by Error Propagation - MIT Press books. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations* (1987), pp. 318–362. URL: https://ieeexplore.ieee.org/document/6302929.

[23] Bank, D., Koenigstein, N. and Giryes, R. Autoencoders. (Mar. 2020). URL: http://arxiv.org/abs/2003.05991.

[24] Gori, M., Monfardini, G. and Scarselli, F. A new model for learning in graph domains. *undefined* 2 (2005), pp. 729–734. DOI: 10.1109/IJCNN.2005.1555942.

[25] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M. and Monfardini, G. The graph neural network model. *IEEE Transactions on Neural Networks* 20 (1 Jan. 2009), pp. 61–80. ISSN: 10459227. DOI: 10.1109/TNN.2008.2005605.

[26] Grover, A. and Leskovec, J. Node2vec: Scalable feature learning for networks. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 13-17-August-2016 (Aug. 2016), pp. 855–864. ISSN: 2154-817X. DOI: 10.1145/2939672.2939754. URL: https://arxiv.org/abs/1607.00653v1.

[27] Perozzi, B., Al-Rfou, R. and Skiena, S. DeepWalk: Online Learning of Social Representations. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Mar. 2014), pp. 701–710. DOI: 10.1145/2623330.2623732. URL: http://arxiv.org/abs/1403.6652.

[28] Hamilton, W. L., Ying, R. and Leskovec, J. Representation Learning on Graphs: Methods and Applications. (Sept. 2017). URL: https://arxiv.org/abs/1709.05584v3.

[29] Hamilton, W. L. Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* 14 (3), pp. 1–159.

[30] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C. and Yu, P. S. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32 (1 Jan. 2019), pp. 4–24. DOI: 10.1109/TNNLS.2020.2978386. URL: http://arxiv.org/abs/1901.00596.

[31] Berto, F. *GitHub - Juju-botu/ICLR2022-OpenReviewData: Crawl & visualize ICLR papers and reviews*. URL: https://github.com/Juju-botu/ICLR2022-OpenReviewData.

[32] Leskovec, J. *CS224W: Machine Learning with Graphs*. 2021. URL: http://snap.stanford.edu/class/cs224w-2020/slides/06-GNN1.pdf.

[33] Oono, K. and Suzuki, T. Graph Neural Networks Exponentially Lose Expressive Power for Node Classification. (May 2019). URL: https://arxiv.org/abs/1905.10947v5.

[34] Cai, C. and Wang, Y. A Note on Over-Smoothing for Graph Neural Networks. (June 2020). URL: https://arxiv.org/abs/2006.13318v1.

[35] Defferrard, M., Bresson, X. and Vandergheynst, P. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. *Advances in Neural Information Processing Systems* (June 2016), pp. 3844–3852. ISSN: 10495258. URL: https://arxiv.org/abs/1606.09375v3.

[36] Kipf, T. N. and Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. *5th International Conference on Learning Representations, ICLR*

2017 - Conference Track Proceedings (Sept. 2016). URL: https://arxiv.org/abs/1609.02907v4.

[37] Wu, F., Zhang, T., Souza, A. H. de, Fifty, C., Yu, T. and Weinberger, K. Q. Simplifying Graph Convolutional Networks. *36th International Conference on Machine Learning, ICML 2019* 2019-June (Feb. 2019), pp. 11884–11894. URL: https://arxiv.org/abs/1902.07153v2.

[38] James, W. *The Principles of Psychology*. Vol. 1. Henry Holt, 1890, pp. 403–404.

[39] Monti, F., Boscaini, D., Masci, J., Rodolà, E., Svoboda, J. and Bronstein, M. M. Geometric deep learning on graphs and manifolds using mixture model CNNs. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017* 2017-January (Nov. 2017), pp. 5425–5434. DOI: 10.1109/CVPR.2017.576. URL: https://arxiv.org/abs/1611.08402v3.

[40] Zhang, J., Shi, X., Xie, J., Ma, H., King, I. and Yeung, D. Y. GaAN: Gated Attention Networks for Learning on Large and Spatiotemporal Graphs. *34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018* 1 (Mar. 2018), pp. 339–349. URL: https://arxiv.org/abs/1803.07294v1.

[41] Battaglia, P., Pascanu, R., Lai, M., Rezende, D. and Kavukcuoglu, K. Interaction Networks for Learning about Objects, Relations and Physics. *Advances in Neural Information Processing Systems* (Dec. 2016), pp. 4509–4517. ISSN: 10495258. URL: https://arxiv.org/abs/1612.00222v1.

[42] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O. and Dahl, G. E. Neural Message Passing for Quantum Chemistry. *34th International Conference on Machine Learning, ICML 2017* 3 (Apr. 2017), pp. 2053–2070. URL: https://arxiv.org/abs/1704.01212v2.

[43] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gulcehre, C., Song, F., Ballard, A., Gilmer, J., Dahl, G., Vaswani, A., Allen, K., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y. and Pascanu, R. Relational inductive biases, deep learning, and graph networks. (June 2018). ISSN: 2331-8422. URL: https://arxiv.org/abs/1806.01261v3.

[44] Murphy, R. L., Srinivasan, B., Ribeiro, B. and Rao, V. Janossy Pooling: Learning Deep Permutation-Invariant Functions for Variable-Size Inputs. *7th International Conference on Learning Representations, ICLR 2019* (Nov. 2018). DOI: 10.48550/arxiv.1811.01900. URL: https://arxiv.org/abs/1811.01900v3.

[45] Seo, Y., Defferrard, M., Vandergheynst, P. and Bresson, X. Structured Sequence Modeling with Graph Convolutional Recurrent Networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 11301 LNCS (Dec. 2016), pp. 362–373. ISSN:

16113349. DOI: 10.48550/arxiv.1612.07659. URL: https://arxiv.org/abs/1612.07659v1.

[46] Yu, B., Yin, H. and Zhu, Z. Spatio-Temporal Graph Convolutional Networks: A Deep Learning Framework for Traffic Forecasting. *IJCAI International Joint Conference on Artificial Intelligence* 2018-July (Sept. 2017), pp. 3634–3640. DOI: 10.24963/ijcai.2018/505. URL: http://arxiv.org/abs/1709.04875.

[47] Chen, J., Wang, X., Xu, ·. X., Chen, J., Wang, X. and Xu, X. GC-LSTM: Graph Convolution Embedded LSTM for Dynamic Link Prediction. *Applied Intelligence* (Dec. 2018), pp. 1–12. ISSN: 15737497. URL: https://arxiv.org/abs/1812.04206v2.

[48] Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., Deng, M. and Li, H. T-GCN: A Temporal Graph ConvolutionalNetwork for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems* 21 (9 Nov. 2018), pp. 3848–3858. DOI: 10.1109/TITS.2019.2935152. URL: http://dx.doi.org/10.1109/TITS.2019.2935152.

[49] Cui, Z., Henrickson, K., Ke, R. and Wang, Y. Traffic Graph Convolutional Recurrent Neural Network: A Deep Learning Framework for Network-Scale Traffic Learning and Forecasting. *IEEE Transactions on Intelligent Transportation Systems* 21 (11 Feb. 2018), pp. 4883–4894. ISSN: 15580016. DOI: 10.48550/arxiv.1802.07007. URL: https://arxiv.org/abs/1802.07007v3.

[50] Pareja, A., Domeniconi, G., Chen, J., Ma, T., Suzumura, T., Kanezashi, H., Kaler, T., Schardl, T. B. and Leiserson, C. E. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. *AAAI 2020 - 34th AAAI Conference on Artificial Intelligence* (Feb. 2019), pp. 5363–5370. ISSN: 2159-5399. DOI: 10.1609/aaai.v34i04.5984. URL: https://arxiv.org/abs/1902.10191v3.

[51] Bishop, C. M. et al. *Neural networks for pattern recognition*. Oxford university press, 1995.

[52] Li, J., Peng, J., Liu, S., Weng, L. and Li, C. TSAM: Temporal Link Prediction in Directed Networks based on Self-Attention Mechanism. (Aug. 2020). URL: https://arxiv.org/abs/2008.10021v1.

[53] Goodfellow, I., Bengio, Y. and Courville, A. *Deep learning*. MIT press, 2016.

[54] Reed, R. and MarksII, R. J. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press, 1999.

[55] Kingma, D. P. and Ba, J. L. Adam: A Method for Stochastic Optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (Dec. 2014). DOI: 10.48550/arxiv.1412.6980. URL: https://arxiv.org/abs/1412.6980v9.

[56] *https://www.python.org/*.

[57] *https://pytorch.org*.

[58] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J. and Chintala, S. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (Dec. 2019). ISSN: 10495258. URL: https://arxiv.org/abs/1912.01703v1.

[59] *https://pytorch-geometric.readthedocs.io*.

[60] Fey, M. and Lenssen, J. E. Fast Graph Representation Learning with PyTorch Geometric. (Mar. 2019). ISSN: 2331-8422. URL: https://arxiv.org/abs/1903.02428v3.

[61] Prechelt, L. Early stopping - But when?: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7700 LECTURE NO (2012), pp. 53–67. ISSN: 16113349. DOI: 10.1007/978-3-642-35289-8_5.

[62] Ba, J. L., Kiros, J. R. and Hinton, G. E. Layer Normalization. (July 2016). DOI: 10.48550/arxiv.1607.06450. URL: https://arxiv.org/abs/1607.06450v1.

APPENDIX A: LIST OF STOCKS

Afarak Group Se
Alandsbanken (A)
Alandsbanken (B)
Amer Sports
Apetit
Aspo
Aspocomp Group
Atria
Biohit (B)
Bittium
Citycon
Comptel
Cramo
Digia
Digitalist Group
Dovre Group
Efore
Elcoteq Se
Elecster (A)
Elisa
Evia
F-secure
Finnair
Finnlines
Fiskars
Fiskars (K)
Fortum
Geosentric
Hkscan (A)
Honkarakenne (B)
Huhtamaki
Ilkka

Incap
Innofactor
Kemira
Keskisuomalainen (A)
Kesko (A)
Kesko (B)
Kesla (A)
Konecranes
Lemminkainen
Marimekko
Martela
Metsa Board (A)
Metsa Board (B)
Metso
Nokia
Nokian Renkaat
Nordea Bank
Norvestia
Olvi (A)
Op Corporate Bank
Oral Hammaslaakarit
Outokumpu
Pkc Group
Ponsse
Poyry
Raisio (K)
Raisio (V)
Ramirent
Rapala Vmc
Rautaruukki
Reka Industrial
Rocla Oy
Saga Furs
Sampo (A)
Sanoma
Solteq
Sponda
Stockmann (A)
Stockmann (B)
Stonesoft

Stora Enso (A)
Stora Enso (R)
Stromsdal
Symphonyeyc Finland
Takoma
Talentum
Tamfelt
Tamfelt
Technopolis
Teleste
Tietoevry
Tiimari
Tulikivi (A)
Turvatiimi
Upm-Kymmene
Uponor
Vaisala (A)
Valoe
Viking Line
Wartsila
Yit