Title

Bitcoin Fraud Detection Using Graph Neural Networks

Author

Dahal, Laxman

Publication Date

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA
Los Angeles

Bitcoin Fraud Detection Using Graph Neural Networks

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Statistics

by

Laxman Dahal

2024

ABSTRACT OF THE THESIS

Bitcoin Fraud Detection Using Graph Neural Networks

by

Laxman Dahal
Master of Science in Statistics
University of California, Los Angeles, 2024
Professor YingNian Wu, Chair

Graph neural network (GNN) is one of the most widely used methods that leverage relational information in the data to learn and make predictions. Fraud detection is a challenging task considering the nature of fraudulent transactions which changes drastically from one case to another as fraudsters often collude to hide their abnormal behavior/features. To this end, GNN has a fitting application because it leverages graph structure to learn relational information to distinguish malicious transactions from legitimate ones. This study implements various GNNs such as graph convolution network (GCN), graph attention network (GAT), and modified GAT to predict fraudulent Bitcoin transactions. It focuses on benchmarking the results of two versions of GAT against GCN to demonstrate the superior predictive power of the attention mechanism. The two versions include conventional GAT and modified GAT, the latter consists of a dynamic attention mechanism. The two versions of the GAT model are also compared in detail. GNN has been used to detect fraud or anomalies for various practical implementations such as financial transactions, credit cards, and customer reviews. However, a detailed study focusing on the two versions of GAT and benchmarking it against GCN has not yet been conducted. We show that GAT has an enhanced ability to

predict fraudulent transactions. The excellent predictive performance of GAT gives a clear indication that it could play a vital role in detecting broader cryptocurrency fraud. Finally, this study discusses the challenges of building an explainable GNN models.

The thesis of Laxman Dahal is approved.

Mark Stephen Handcock

Frederic R Paik Schoenberg

YingNian Wu, Committee Chair

University of California, Los Angeles

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

1.1 Background

In recent years, Graph neural networks (GNNs) have become one of the emerging research areas in the machine learning (ML) domain. From molecular fingerprinting (Jumper et al., 2021) to predicting a source of misinformation in a social network (Monti et al., 2019), GNNs have been implemented to solve a host of problems involving networks or graphs. GNNs are designed to operate on networks, making them readily applicable to real-life applications such as disease pathways, transportation networks, social misinformation web, and financial transaction networks. These networks are incredibly complex to be accurately represented by tabular data in Euclidean space. Hence, it is challenging for conventional deep learning models such as convolutional neural networks (CNNs) to learn and make predictions adequately. The inherent unstructured nature of these problems makes them suitable to be represented as a graph. Additionally, a graph can accommodate rich relational information (e.g., the strength and direction of connections) between its entities. GNN leverages such relational information to make inferences. This indicates that any GNN or graph-based statistical method should, at the very least, be able to operate on a non-Euclidean data structure. To this end, GNNs are mathematically formulated to explicitly elucidate complex relationships and interactions between entities in a network. The underlying objective of GNN is to use representation learning to automatically learn features based on the arbitrary topological structure of a graph. Several studies across various fields (Jumper et al., 2021; Zhou et al., 2020a; Fan et al., 2019; Hamilton et al., 2017a) have demonstrated that state-of-

the-art GNNs can be developed by exploiting arbitrary topological structure and relational information contained within a graph.

Although graphs are becoming the new frontier of deep learning, attempts to develop a neural network architecture for graphs date back to the late 1990s. Several studies (Sperduti and Starita, 1997; Frasconi et al., 1998) proposed modified versions of recursive neural networks (RNN) with a focus on graph structure such as directed acyclic graphs. Although novel at the time, these methods were plagued by convergence issues, limited expressivity, and representation inabilities. In 2010, the classic multilayer feedforward neural network trained with the back-propagation algorithm (LeCun et al., 1998) was extended to perform supervised learning tasks in a class of graphs such as cyclic/acyclic, and directed/undirected graphs (Micheli, 2009; Scarselli et al., 2008). These methods alleviated some of the limitations of the previous models, but their reliance on iteration to attain convergence limited their extendability and scalability. However, the paradigm-shifting success of deep learning models such as convolutional neural network (CNN) (LeCun et al., 2015) renewed the need to develop GNN. Thus, in a certain sense, the advancement in GNN architectures can be attributed to the simple fact that CNN is designed to operate on one-dimensional (1D) (e.g., text or speech) or two-dimensional (e.g., images) grid data. For tasks such as image classification and semantic segmentation, CNN transforms the grid or sequence data to Euclidean space before passing it through the hidden convolution layer and looping mechanism to learn and make predictions. Specifically, CNN aims to predict spatially localized features such as identifying handwritten digits or completing sentences in a paragraph. The essential components of CNNs include local connections, shared weights, and multiple hidden layers. In literature, several researchers have attempted to generalize CNNs for graph data without much success. The challenge is developing convolutional filters and pooling operators that can be implemented in both the Euclidean and non-Euclindean domains. Hence, the first successful proposal of a graph-based neural network architecture in the form of a graph convolutional network (GCN) (Kipf and Welling, 2016) took advantage of the notion

that a computational graph (and subsequently a GNN) can be defined by a node's neighbors. Specifically, the key idea in GCN is generating node embedding by aggregating local neighborhood information and propagating that information across graphs to compute node features.

1.2 Machine Learning for Graphs

A graph is a visual representation of the mathematical structure used to model the pairwise (directed or undirected) relationship between two entities or objects. Mathematically, a graph is defined as $G = (V, R, E)$ where $v_i \in V$ are the nodes and $(v_i, r, v_j) \in E$ are the edges with relation type $r \in R$. Figure 1.1 highlights two classes of heterogeneous graphs: directed and undirected. At a minimum, two key components (nodes and edges) are required to define a graph (i.e., $G = (V, E)$), while additional features such as relationship type (R) are optional. In Figure 1.1, the circled numbers represent the vertices (V) that are connected by edges (E) (black line). The term "vertices" is synonymous with "nodes" and is hereafter used interchangeably. As highlighted in the figure, the edges (or equivalently links) can be directed or undirected. At its core, a graph simply defines the relationship and interaction between nodes.

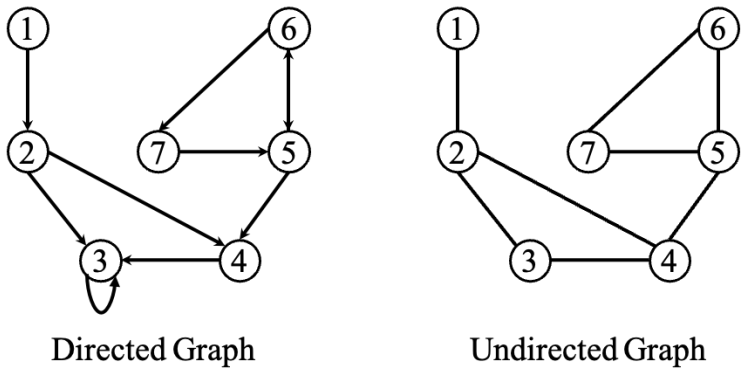


Figure 1.1: Graphical representation of directed (left) and undirected graph (right).

In graph-based ML, the downstream tasks can be broadly classified into three classes as shown in Figure 1.2. These tasks are described as follows:

1. **Node-level tasks** aim to learn node-level features and solve the problem at an individual node level. An example of a node-level task is node classification where the objective could be to determine if a transaction is fraudulent. The node-level task is the most granular task possible in a graph.

2. **Edge-level tasks** focus on the connection and relationship between two nodes. Some of the edge-level tasks include side-effect prediction of two drugs, recommender system, etc.

3. **Graph-level tasks** involve an entire graph (or subgraph). Several types of actions such as graph classification, clustering, and new graph generation can be performed at the graph level.

1.2.1 Traditional ML Tasks

Traditional ML models such as linear regression, random forests, support vector machines (SVM), and neural networks can handle all three hierarchies of tasks on a graph. However, these models require a rich set of features for them to achieve good performance. To distinguish from graph-based ML, the term "traditional" ML is used to represent any ML algorithm that is not designed to inherently handle geometric data structure. In a traditional ML pipeline, designing features and obtaining data to train and test the model is integral to the success of the model. In fact, feature engineering is often leveraged to push the mathematical limitations of an ML model and achieve high-level performance. Feature engineering on graphs, however, has an added layer of complexity. In addition to capturing the relational information between nodes, the features also need to correctly describe the topological pattern of a network. Based on different ML tasks highlighted in Figure 1.2, the

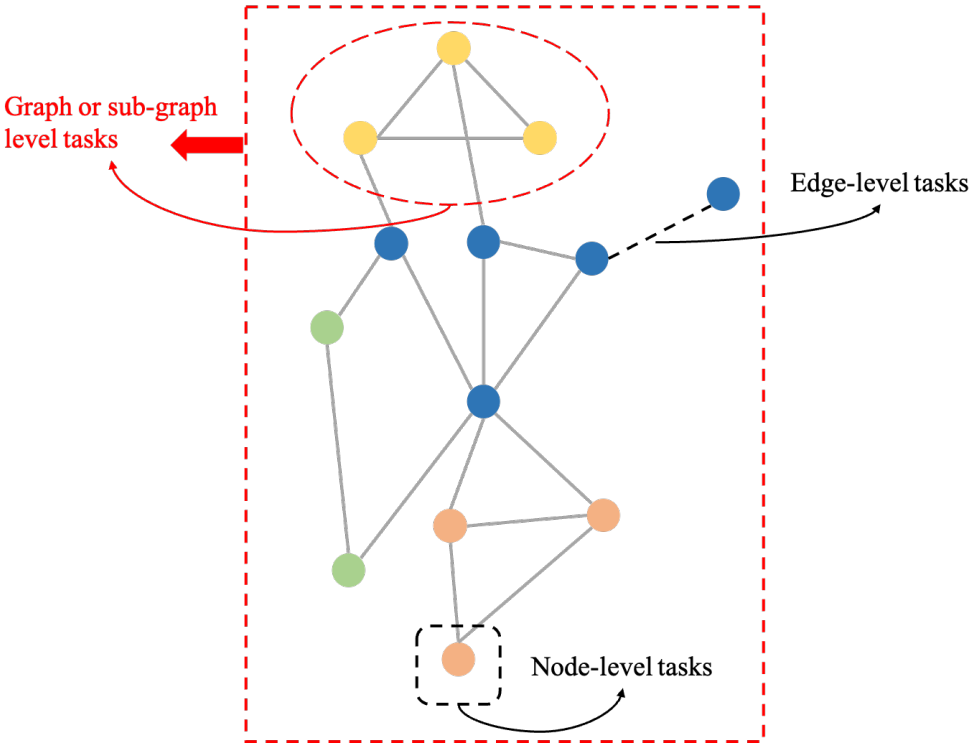


Figure 1.2: Three types of hierarchies in downstream ML tasks on graphs.

following sub-sections give an insight into feature engineering for traditional ML tasks on graphs.

Node-Level features

For node-level prediction, ML models need node features that can capture the importance of a node in the graph as well as the topological structure around the node. The overall goal of the node-level features is to characterize the relevance and significance of a node in the network. Some of the approaches to extract node-level features are listed below:

1. **Node degree:** The degree (k_v) of a node (say v) is simply the number of edges connected to the node. It is one of the most basic node-level features that captures the network structure without capturing the importance of a node or the neighboring nodes.

2. **Node centrality:** Node centrality is intended to model the importance of a node to the graph. Various node centrality methods are available such as eigenvector centrality, betweenness centrality, and closeness centrality.

3. **Clustering coefficient:** A clustering coefficient is a measure of the degree to which nodes in a graph tend to cluster together. It is given by $e_v = \frac{\#\text{of edges among neighboring node}}{\binom{k_v}{2}}$ where k_v is the degree of node, v, and $e_v \in [0, 1]$. The clustering coefficient (e_v) measures the connectivity of $v's$ neighborhood nodes. For instance, $e_v = 1$ signifies that every neighboring node (of node v) in a graph or subgraph is connected to each other while $e_v = 0$ implies that none of the neighboring nodes are connected to each other. This feature is useful to capture a one-degree neighborhood around a given node.

4. **Graphlets** Graphlets are small subgraphs that are intended to describe the network structure around the node of interest. Unlike the clustering coefficient, graphlets are

not limited to the one-degree neighborhood and can be generalized for a specified subgraph.

The non-exhaustive list highlights some of the potential node-level features a traditional ML model could leverage for node-level tasks such as node classification. For instance, node degree, node centrality, and clustering coefficient could be used to detect the popularity of an influence on social media platforms.

Link-level features

Link-level prediction tasks require link-level features that define relational properties between two nodes. The objective of the link-level task is to predict new links based on the existing links in the network. Mapping out citation networks is an excellent example of link-level tasks where the goal is to predict the missing citation links. The three most common link-level features are explained below:

1. **Distance-based features:** There are different variations of distance-based features- all of which are based on the premise of computing distance between two nodes. The shortest path distance is one of the most common metrics used as the link-level feature.

2. **Local neighborhood overlap:** Local neighborhood overlap captures the number of neighboring nodes shared between two nodes v_1 and v_2. Several metrics are available to quantify the local neighborhood overlap: common neighbors, Jaccard's coefficient (Niwattanakul et al., 2013), and the Adamic-Adar index (Adamic and Adar, 2003).

3. **Global neighborhood overlap:** One of the limitations of the local neighborhood overlap is that the metric is always zero if two nodes do not have any neighbors in common. To overcome this limitation, global neighborhood overlap accounts for the entire graph. One of the most popular global neighborhood features is the Katz index (Katz, 1953) which counts the number of all the possible walks between a given pair of

nodes. The Katz index is used to score two nodes which is subsequently used to make link predictions.

Graph-level features

For graph-level tasks such as graph generation, graph-level features that effectively encompass the structure of an entire network are essential to make predictions. Since a graph can be complex, arbitrary, and sometimes ambiguous, distance- or centrality-based metrics (as widely used as node- and link-level features) are inadequate in capturing the structure and function of a graph or subgraph. At a graph level, feature vectors are designed as kernels. For instance, a kernel can simply be defined as $K(G, G') = \phi(G)^T \cdot \phi(G')$ where $\phi(\cdot)$ is the feature representation of the graph. The kernel, $K(G, G') \in \mathbb{R}$, measures the similarity between the data in the graph. Once a kernel is defined, off-the-shelf ML models such as kernel SVM can directly be used to make predictions.

1. **Graphlet kernel:** Graphlet kernel draws its motivations from the idea of bag-of-words (McCallum et al., 1998; Joachims, 1998). The bag-of-words uses the word count as a feature for documents with no regard to the order of the words. In graph theory, this concept can be generalized as the bag-of-graphlets since the underlying idea is counting the number of graphlets. In graphlet kernel, a list of graphlets of size k is assembled to compute the kernel. One of the limitations of the graphlet kernel is that counting the $k-$sized graphlets for a graph with size n is computationally expensive since the enumeration takes n^k computational complexity.

2. **Weisfeiler-Lehman kernel:** To address the computational limitations of graphlet kernel, Weisfeiler-Lehman (WL) kernel (Shervashidze et al., 2011) was proposed to design an efficient graph feature descriptor $\phi(G)$. The WL kernel is a generalization of the bag-of-node-degrees that leverage $K-$hop neighborhood information to iteratively enrich node information in the kernel. Specifically, the WL kernel applies a $K-$step

color refinement algorithm (often hash maps) to enrich node colors. To capture different $K-$hop neighborhood structures, nodes with similar colors are grouped together. Finally, the idea of a bag of colors is implemented to represent the graph. The WL kernel is computationally efficient as the time complexity is linear in the number of edges for each iteration step.

Several other graph kernels such as random-walk kernel (Vishwanathan et al., 2010) and shortest-path graph kernel (Borgwardt and Kriegel, 2005) have also been proposed in the literature but are not presented here for brevity. For the scope of this study, if one wanted to implement traditional ML algorithms, developing node- and link-level features would be the most relevant.

1.3 Node Embedding

The Section 1.2.1 discussed feature engineering at three different levels for various downstream ML tasks such as node classification, link prediction, and new graph generation. Figure 1.3 highlights the overview of the workflow. One of the key components of traditional ML tasks is the need for feature engineering and feature design to capture the various aspects of a graph such as the topological structure of a network, the importance of a node in the network, etc. The "modern" graph ML leverages graph representation learning which alleviates the need to develop graph- and task-specific features. As demonstrated in Figure 1.3, for a given graph, representation learning is implemented to compute a low-dimensional feature vector for each node in the graph. The low-dimensional vector is subsequently used as the input for the downstream ML tasks. In the subsequent sections, several widely used node embedding algorithms are discussed.

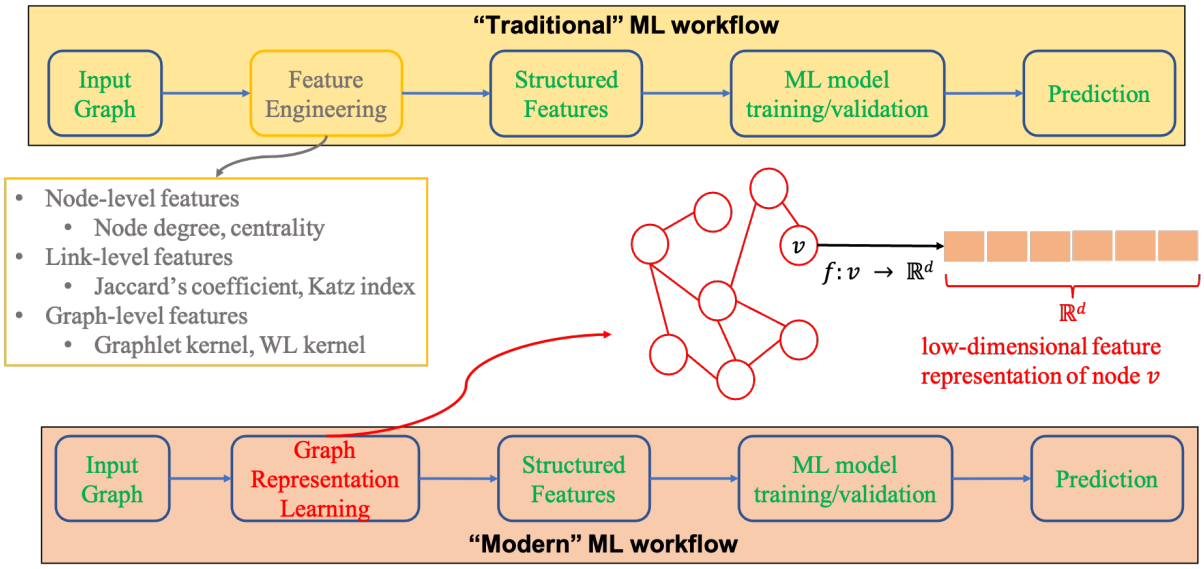


Figure 1.3: Comparison of a traditional and modern ML workflow on a graph. In the traditional workflow, extensive feature engineering is required but in modern workflow graph representation learning is leveraged.

1.3.1 Encoder and Decoder

Within the node embedding space, the encoder and decoder framework (Hamilton et al., 2017b) is proposed as a unified workflow that capitalizes on the idea of an encoder function (Equation 1.1a) that maps each node to a low-dimensional vector, and a decoder function (Equation 1.1b that reconstructs the graph from a learned encoder. Generally, the encoder and decoder are formulated as

$$\text{ENC} : \text{ENC}(v) \rightarrow \mathbf{z_v} \in \mathbb{R}^d \tag{1.1a}$$

$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(z_u, z_v) \approx s_{\mathcal{G}}(u, v) \tag{1.1b}$$

where $\text{ENC}(v)$ embeds node v of graph $\mathcal{G}$ into a $d-$dimensional space and a pairwise decoder $\text{DEC}(z_u, z_v)$ computes a similarity measure which quantifies the similarity of two nodes (u and v) in the original graph ($\mathcal{G}$). The goal is to optimize the encoder and decoder functions such that similarities between u and v in the original graph (i.e. $s_{\mathcal{G}}(u, v)$) is adequately

reconstructed by $\text{DEC}(z_u, z_v)$ with minimal loss. The loss function is represented as

$$\mathcal{L} = \sum_{(u,v)\in\mathcal{D}} \ell\left(\text{DEC}(z_u, z_v), s_{\mathcal{G}}(u, v)\right) \tag{1.2}$$

where $\mathcal{D}$ is a set of training node pairs. Once the encoder-decoder engine is optimized, the encoder can be used to generate embeddings, which can then be directly used as the feature inputs for the downstream ML tasks. It is important to note that the encoder-decoder framework is unsupervised or self-supervised learning that relies on neither the node labels nor the node features. Additionally, node embedding is task-independent and can be used for any ML task.

1.3.2 Random Walk-Based Approaches

The encoder-decoder system presented in Section 1.3.1 is a generic framework for node embeddings. Conceptually, the simplest encoder-decoder function could be a simple embedding function (also sometimes known as "shallow" embedding) where the encoder is just a lookup function that returns a unique embedding vector that each node is assigned. However, with the advances in computational capacity, several studies have proposed a stochastic measure of node similarity based on the idea of random walks (Perozzi et al., 2014; Grover and Leskovec, 2016). Similar to shallow encoding, the goal is to learn the embedded vector representation of node u (i.e., z_u) by leveraging the predicted probability of visiting node v on random walks starting from node u. Equation 1.3 outlines the loss function random walk-based approaches aim to minimize by using a random walk strategy, R. that makes N_R random walks.

$$\mathcal{L} = \sum_{u\in\mathcal{V}} \sum_{v\in N_R(u)} -\log\left(\frac{\exp(z_u^\mathsf{T} z_v)}{\sum_{n\in V}\exp(z_u^\mathsf{T} z_n)}\right) \tag{1.3}$$

DeepWalk and *node2vec* are the two most popular random walk-based embedding approaches but use different optimizations and approximations to compute the loss in Equation 1.3. DeepWalk employs a "hierarchical softmax" technique to compute the normalizing

factor, using a binary-tree structure to accelerate the computation. In contrast, node2vec approximates Equation 1.3 using "negative sampling" instead of normalizing over the full vertex set. In other words, node2vec approximates the normalizing factor using a set of random "negative samples" and allows for a flexible definition of random walks.

1.4 Fraud Detection Using GNN: A Survey

Fraud is almost as old as the World Wide Web. It is a decades-old problem. There are different shapes and forms of fraudulent activities prevalent in society. For generalization, fraud can be grouped under the umbrella term anomalies. Anomalies appear in virtually all sectors- including but not limited to fabricated citation networks, spam email detection, controversy detection, fake news detection, source of cyberbullying, rumor detection, and source of fake tweets. It is not an exaggeration to say that virtually every ML algorithm ever proposed has been implemented to detect fraud or anomaly. Anomaly detection is a unique problem that continues to evolve over time. The following three factors make anomaly detection a challenging problem: 1) anomalies, by definition, are not a normal occurrence which signifies that the dataset to train and test the model tends to be highly imbalanced, 2) the nature, scope, and scale of the fraudulent activities evolve as fraudsters leverage technology to camouflage themselves better and develop sophisticated fraud schemes, and 3) the point of origin of the fraud if often unique from one fraud to another which makes it difficult to pinpoint the root node and learn from its corresponding attributes. Although the scope of this study is financial fraud, the remainder of this section presents a non-exhaustive survey of state-of-the-art methods developed to detect fraud across various sectors and benchmark datasets. Fundamentally, the studies presented below attempt to solve the three challenges outlined above.

Financial fraud detection is an active area of research with thousands of research publications every year. The vast population of research is grouped into two broader groups: GCN

and GAT. In the past decade or so there has been substantial development in graph-based ML algorithms which can be roughly grouped into two groups based on the development of GCN and GAT. It is indisputable that GCN and GAT helped lead the advancement of GNN methods. To this end, Table 1.1 highlights the non-exhaustive, hand-curated list of recent studies that have leveraged GNN methods to detect fraud in financial and social networks. Specifically, Lu and Li (2020) proposed Graph-aware Co-Attention Networks (GCAN) to predict a fake tweet based on the retweets and the features of the account retweeting. GCAN makes use of Gating Recurrent Units (GRU) and 1-D CNN to learn the sequential correlation of retweets. In any modern internet platform, a recommender system is the backbone and often determines the success of any e-commerce or digital platform including social media. Identifying and removing illicit actors in the network is crucial to developing a responsible recommender system. To address this, Zhang et al. (2020) proposed a robust recommender system that implements GCN to eliminate the fraudulent reviews from the dataset before training the recommender system. Similarly, some studies (Wang et al., 2019) have researched the impact of the evolution of licit customers' financial behavior and how that can be leveraged to detect fraudulent behaviors. The financial transactional data often exhibits multifaceted information as the customer's financial behaviors evolve over time. To extract a richer representation, the proposed study (Wang et al., 2019) implemented a large multi-view network through a hierarchical attention mechanism to better correlate different neighbors and different views. By expanding the labeled data through social relations, the authors were able to achieve comparable accuracy to the state-of-the-art methods when implemented on the Alipay dataset.

More recently, researchers have started to explore various ways to enhance the established GCN and GAT architecture. For instance, Dou et al. (2020) did extensive work on detecting camouflaged fraudsters and proposed the CARE-GCN algorithm. Specifically, they developed a label-aware similarity measure to identify camouflage at the feature level (i.e., node level) and at the link level. Subsequently, they implemented RL to determine the optimal

Table 1.1: Summary of the GNN algorithms developed to detect and predict frauds

GNN Class	Algorithm	Dataset	Application	Reference
GCN	GRU + GCN	Twitter	Detect the source of fake tweet	GCAN (Lu and Li, 2020)
	GCN	Yelp, Amazon	Robust recommender system that account for fake reviews	GraphRfi (Zhang et al., 2020)
	RL + GNN	Yelp, Amazon	Detect camouflaged fraudsters	CARE-GNN (Dou et al., 2020)
GAT	Hierarchical GAT	Alipay	A multi-view attention mechanism to predict financial fraud	SemiGNN (Wang et al., 2019)
	LSTM + GNN	Alipay	Predict credit risk	TemGNN (Wang et al., 2021)
	Multi-attention GAT	Alibaba	Predict credit card defaulter	MAHINDER (Zhong et al., 2020)
	Gated Temporal GAT	Yelp, Amazon	Detect credit card fraud	GTAN (Xiang et al., 2023)

amounts of neighbors to be selected across different relations which are ultimately aggregated to detect a camouflaged fraudster. At the time of the publication, the CARE-GNN was at the top of the leaderboard for two benchmarks- Yelp and Amazon datasets. A standard GNN model can also be advanced by utilizing the temporal properties of the network as proposed by Xiang et al. (2023). The authors proposed a semi-supervised model that is designed to work on the temporal transaction graph. The messages among the temporal nodes are passed through the gated attention network which learns the transaction representation and helps model the fraud patterns through risk propagation. At the time of this writing, the GTAN is the top-ranked algorithm for two fraud detection benchmarks (Yelp and Amazon datasets).

1.5 Objectives

The primary objective of this study is to implement various GNN models to investigate fraud detection in Bitcoin transactions. Specifically, we aim to implement GCN, GAT, and

modified GAT to 1) detect fraudulent transactions and 2) evaluate if the attention mechanism in GAT boosts the performance. In a financial transaction network, fraud detection is often represented as a node-level task where account holders are represented as a node and the goal is to determine if a given account holder is a fraud or not. Equivalently, the same fraud detection can also be framed as an edge-level prediction problem by representing the transaction as an edge between two nodes (customers). The flexibility in problem formulation makes GNN extremely adaptable and generalizable. In fact, the definition of learning and prediction tasks depends on the problem and consequently network representation. If the fraud detection problem is framed as the node classification task, GNN is designed to learn graph structure at the node level in addition to learning the node-level features. Similarly, if the problem is outlined as the edge classification, the graph structure is learned based on the pair of nodes.

1.6 Organization

The remainder of the thesis is organized as follows. In Chapter 2, a spectrum of GNN-based architectures such as GCN, GAT, and modified GAT are presented. The chapter gives a detailed mathematical background into the aforementioned architectures and a guide on how to build a GNN. In Chapter 3, the three GNN models are implemented to detect fraudulent Bitcoin transactions. A detailed performance evaluation of the models is also presented while comparing the three models. Finally, Chapter 4 summarizes findings, discusses limitations, and provides recommendations for potential future studies.

CHAPTER 2

GNN Architectures

2.1 Basics of Graph Neural Networks

Some of the fundamental concepts of GNN such as node embedding (Section 1.3), and feature representation (Section 1.2.1) have been previously discussed. This section presents a few additional theoretical underpinnings that distinguish GNN from CNN. Some common concepts such as stochastic gradient descent, forward-propagation, back-propagation, regularization, and activation functions are not discussed in detail for brevity.

2.1.1 Permutation Invariance and Equivariance

Permutation invariance/equivariance is one of the unique characteristics of GNN that distinguishes it from other deep learning methods such as CNN. GNNs are inherently invariant to permutation because the graph representation of a network will always be the same no matter how the nodes are ordered. For instance, a network with N nodes will have $N!$ permutations or order plans. The permutation invariant property implies that just learning one function (say, f) is sufficient to map all $N!$ variations of order plans. Similarly, for node representation, GNN is permutation equivariant which signifies that for different order plans, the vector representation of the node at the same position is always the same. GNNs consist of multiple permutation invariant/equivariant functions across various layers which ensures that the model will produce the same output regardless of the order of nodes in the input graph. This property is essentially what makes GNN the perfect fit for unstructured and

unordered data. Other popular models such as CNN are not permutation invariant as they produce different results for the same inputs but with different orders. For instance, CNN utilizes a pooling operation, which is often a fixed dimension (e.g., 3×3 operator) that slides across the feature maps to summarize the feature representation. The pooling operation is not permutation invariant/equivariant, as the output would be totally different if the same image is rotated.

2.1.2 Neighborhood Aggregation

At its core, the mechanism by which the information is aggregated and propagated is what determines if a neural network is permutation invariant/equivariant. In a seminal work on GCN, Kipf and Welling (2016) proposed a key concept of layer-wise propagation rule motivated by the first-order approximation of localized spectral filter on graph (Hammond et al., 2011). The authors demonstrated that by stacking multiple K-localized convolutional layers, rich information can be aggregated, propagated, and transformed to ultimately develop a multi-layer GCN model. Figure 2.1 gives a graphical illustration of a simple GNN model architecture. The input graph is an undirected graph with six nodes where node A is the target node. The primary objective of neighborhood aggregation is to compute node features (of the target node A in this example) through node embeddings based on the local network neighborhood. The node embeddings can be computed using different aggregation approaches across the layers. One of the simplest aggregation approaches is to average the messages from neighbors and propagate them to the downstream layers. Various aggregation operators/functions such as $min(\cdot)$, $max(\cdot)$, and $mean(\cdot)$ can be used to combine the information. It is important to note that the aggregation operator needs to be permutation invariant, for the ordering of h_A and h_C should not matter in the third NN at the 0^{th} convolution layer.

In the Figure 2.1, the two-layer convolution is essentially node A's computational graph. Unique from classical CNN, in GNN, every node gets to define its own computation graph

based on its neighborhood. Ultimately, the idea of neighborhood aggregation is what enables GNN to extract such rich and comprehensive information from spatial graphs.

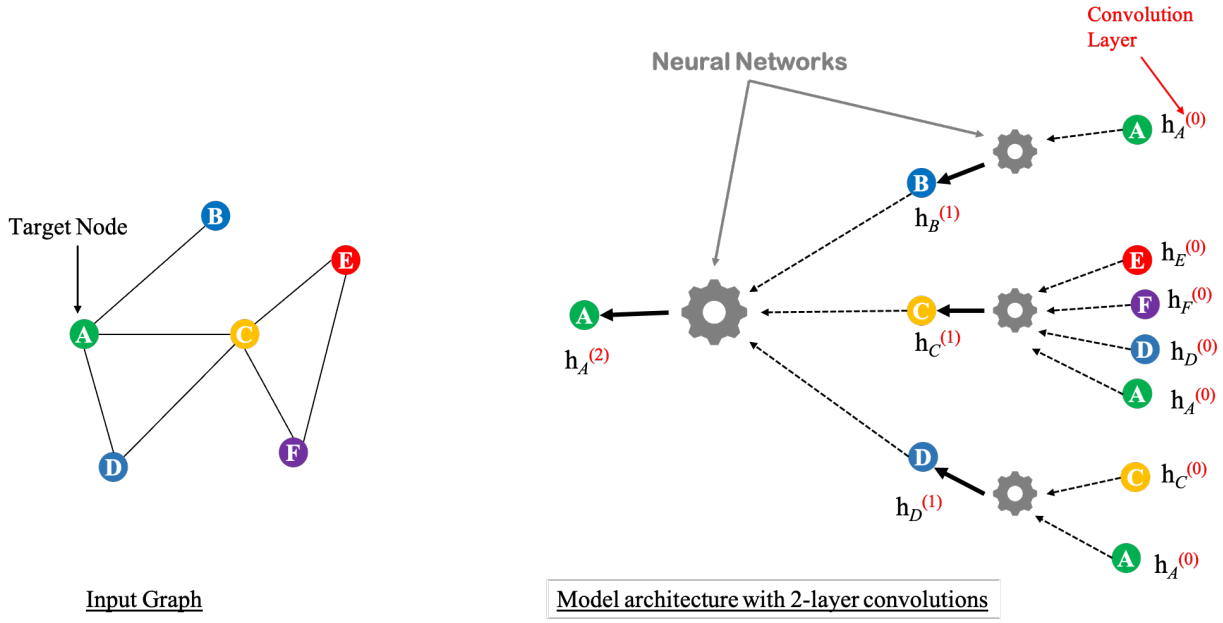


Figure 2.1: Overview of the model architecture with 2-layer convolutions.

2.2 Graph Convolution Network

Graph convolution network (GCN) is the first of its kind to generalize well-established neural network architectures such as RNN or CNN to work on spectral graphs that are arbitrarily structured. Kipf and Welling (2016) demonstrated that by leveraging simple yet robust simplifications of spectral graph convolutions, GCN can be used for fast, scalable, and accurate semi-supervised classification tasks in a graph. A GCN is a neural network model that is built by stacking multiple spectral graph convolutions (as shown in Figure 2.1), each of which is followed by a point-wise nonlinearity. Figure 2.2 shows a graphical representation of a multi-layer GCN with filters.

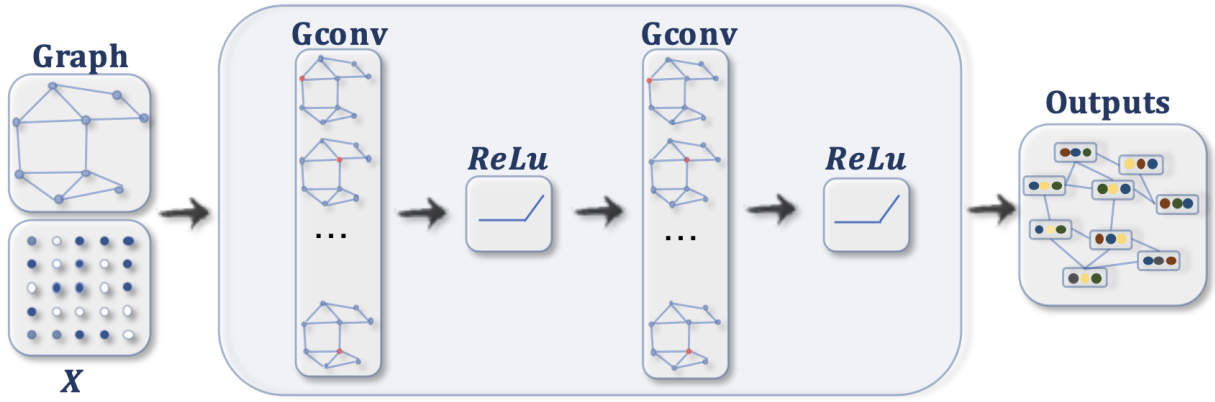


Figure 2.2: Multi-layer GCN with multiple spectral graph convolution. Each convolution layer encapsulates each node's hidden representation through the computational graph by aggregating feature information from its neighbors. Each subsequent convolution is followed by a nonlinear transformation such as ReLu. Source: Wu et al. (2020).

2.2.1 Key Components of a GNN Layer

A single GNN layer within the GCN model is comprised of two primary components 1) message passing or transformation, and 2) aggregation. A GNN layer compresses a set of vectors into a single vector by encoding the information in a neighboring node and self-node to generate the output embedding which is then aggregated, transformed, and propagated to another layer. A typical message computation could look as follows:

$$\mathbf{m}_u^{(l)} = \text{MSG}^l\left(\mathbf{h}_u^{(l-1)}\right) \tag{2.1}$$

where MSG is the message computation function, $\mathbf{m}_u^{(l)}$ is the message of node u, and $\mathbf{h}_u^{(l-1)}$ is the input node embedding. The message computation is intended to encapsulate the message which will be sent to the other nodes. A very simple example of a message computation function is a linear layer (as shown in Equation 2.2) where $\mathbf{W}^{(l)}$ is the weight matrix.

$$\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)} \tag{2.2}$$

The other key component of any given GNN layer is message aggregation. As highlighted in Figure 2.1, the message aggregator function for each node aggregates the messages from the nodes's neighbors. The aggregator function can be formulated as:

$$\mathbf{h}_v^{(l)} = \text{AGG}^{(l)}\left(\mathbf{m}_u^{(l)}, u \in N(v)\right) \tag{2.3}$$

where the AGG() function can be a common aggregation operation such as summation, average, maximum, minimum, or any other distance-based metrics as described in Chapter 1. One of the issues in the aggregator function as described in Equation 2.3 is that the information from node v itself could get lost. This issue can be circumvented by concatenating self-node information to the node embedding as shown highlighted in Equation 2.4

$$\mathbf{h}_v^{(l)} = \text{CONCAT}\left(\text{AGG}^{(l)}\left(\mathbf{m}_u^{(l)}, u \in N(v)\right), \mathbf{m}_v^{(l)}\right) \tag{2.4}$$

where $\mathbf{m}_v^{(l)}$ is the self information of node v computed using Equation 2.2. Once the node-level message is computed and aggregated, the nonlinear activation function (e.g., ReLu, Sigmoid) is used to add expressiveness. The activation function is often written as $\sigma(\cdot)$.

2.2.2 Classical GCN Layer

In the notations that have been described so far, a classical GCN layer as defined in Kipf and Welling (2016), can be mathematically represented as follows

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \mathbf{W}^{(l)} \frac{\mathbf{h}_u^{(l-1)}}{|N(v)|}\right) \tag{2.5}$$

where the message is normalized by some normalization factor which is then aggregated by taking the sum across all neighbors including the messages from self edges. The output is the embedding of node v at layer l. The GCN layer outlined in Equation 2.5 essentially builds a graph-based neural network model, say $f(\cdot)$, that encodes the embeddings. The formulation shown in Equation 2.5, shows the embedding of the l^{th} layer computes using the embedding from the $(l-1)^{th}$ layer.

Alternatively, for the sake of defining the algorithm of the GNN layer ($f(X, A)$), Equation 2.5 can be re-formulated in a more generic form as

$$f(H^{(1)}, A) = \sigma\left(AH^{(1)}W^{(1)}\right) \tag{2.6a}$$

$$f(H^{(1)}, A) = \sigma\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^{(1)}W^{(1)}\right) \tag{2.6b}$$

where $\hat{A} = A + I_N$ is the adjacency matrix of the graph $\mathcal{G}$ with added self-connections, I_N is the identity matrix, $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$, $W^{(l)}$ is the learnable weights, $\sigma(\cdot)$ is an activation function. Additionally, $H^{(l)} \in \mathbb{R}^{N \times D}$ is the embedded feature matrix at the l^{th} layer, $H^{(0)} = X$ is the input feature matrix, $H^{(L)} = Z$ is the output feature matrix, and L is the number of layers. Algorithm 1 outlines the algorithm of the graph convolution layer. The seemingly simple yet noble idea of self-connection and symmetric normalization is what makes GCN so powerful because the symmetric normalization (of A) enables matrix multiplication (e.g., Equation 2.6a) without scaling the feature vectors.

Algorithm 1 Graph Convolutional Layer

Data: Graph adjacency matrix A, Feature matrix X, Learnable weight matrix W, Activation function σ

Result: Output feature matrix H

Input : Graph adjacency matrix A, Feature matrix X, Learnable weight matrix W, Activation function σ

Output: Output feature matrix H

1 $\hat{D}^{-\frac{1}{2}}$ = Diagonal degree matrix of A with elements $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$ $\tilde{A} = A + I_N$; `// add self-connections`

2 $\hat{A} = \hat{D}^{-\frac{1}{2}}\tilde{A}\hat{D}^{-\frac{1}{2}}$; `// symmetric normalization`

3 $H = \sigma(\hat{A} \cdot X \cdot W)$

4 **return** H

2.2.3 Build Practical GNN Model

Like other neural network models, GNN can easily be enhanced by stacking convolutional layers. Since the fundamental idea of GNN is rooted in the neural network framework, modern deep learning modules such as batch normalization, dropout, attention/gating, skip connection, and pooling, can be easily included in the model. Figure 2.3 outlines the overall implementation workflow of a GNN model.

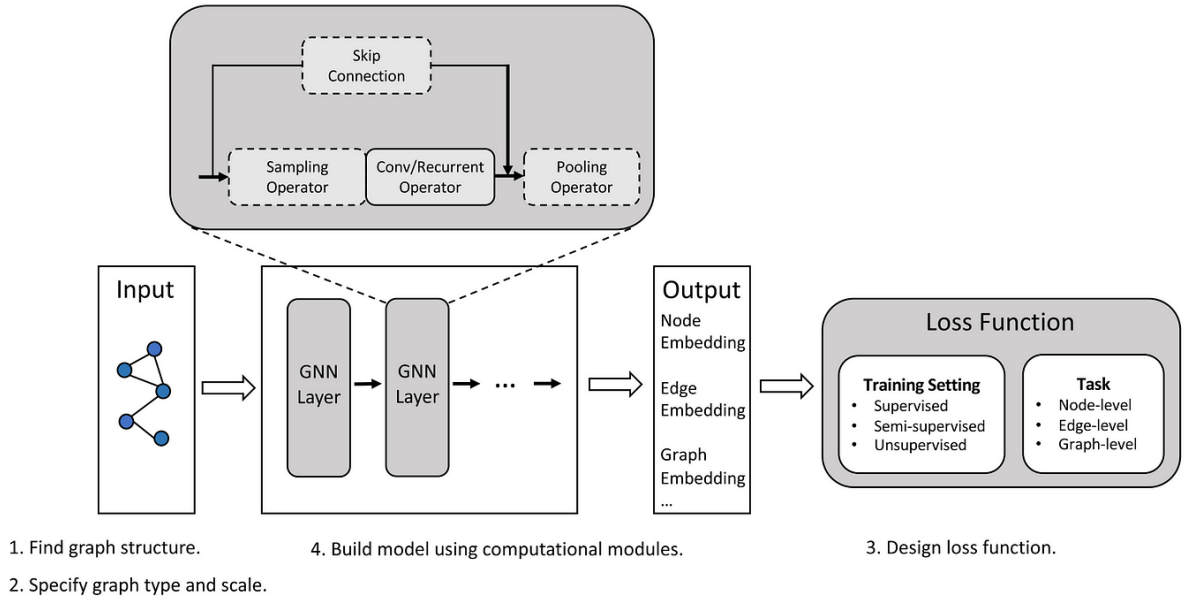


Figure 2.3: Overview of the model architecture with 2-layer convolutions. Source: Zhou et al. (2020b).

In this study, a classical two-layer GCN is implemented with batch normalization but no dropout. The results obtained from the model are presented and compared against other graph-based models in Chapter 3.

2.3 Graph Attention Network

One of the major limitations of GCN is that it applies a uniform convolution operation to all neighbors at any given convolutional layer. In modern deep learning frameworks, attention mechanisms have become a gold standard, especially in many sequence-based tasks (Bahdanau et al., 2014). Inspired by the cognitive attention of human beings, the attention mechanism implicitly determines the neighboring nodes that the node of interest should pay the most attention to. Graph attention networks (GATs) (Veličković et al., 2017) represent a significant advancement in the real of graph-based deep learning. The use of masked self-attentional layers addresses the shortcomings of conventional GCNs and results in a novel GAT architecture that is computationally efficient and does not require spectral features of the entire graph upfront.

2.3.1 Graph Attention Layer

The overarching theoretical and practical underpinning of GATs is to focus on the most relevant parts of the input to make decisions. In the spectral graph domain, *self-attention* is referred to as the node embedding computed to represent a computational graph of the target node through the attention mechanism. In the seminal work on transformers, Vaswani et al. (2017) demonstrated that *self-attention* can significantly improve the performance of sequence-based tasks and construct powerful state-of-the-art deep learning models.

Following the mathematical formulation of GCNs as shown in Equation 2.5, a typical GAT layer can be formulated as

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right) \tag{2.7}$$

where α_{uv} is the weight for every neighbor u of node v. The weight is an implicit way of controlling how attention node V should give its neighboring nodes.

To define a graph attention layer, let N be the total number of nodes in a graph or a

network and F be the number of features in each node. Let us define the input to each graph attentional layer as a set of node features: $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, \ldots, \overrightarrow{h_N}\}$, $\overrightarrow{h_i} \in R^F$. This results in the output feature representation in the form of F': $\mathbf{h'} = \{\overrightarrow{h'_1}, \overrightarrow{h'_2}, \ldots, \overrightarrow{h'_N}\}$, where $\overrightarrow{h'_i} \in \mathbb{R}^{F'}$.

To retain sufficient expressive power, as a pre-processing step, a shared linear transformation, parametrized by a weight matrix, $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is applied to every node. Following the learnable linear transformation, a shared *self-attention* mechanism (α) is implemented on the nodes to compute attention coefficients as shown in Equation 2.8.

$$a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R} \tag{2.8a}$$

$$e_{ij} = a(\mathbf{W_l}\overrightarrow{h_i}, \mathbf{W_r}\overrightarrow{h_j}) \tag{2.8b}$$

The attention coefficients indicate the importance of node $j's$ features to node i. In the baseline formulation, the self-attention would allow every node to attend to all other neighboring nodes. However, to only focus on the important part of the graph structure, masked attention is implemented. In masked attention, attention coefficients are only computed in some neighborhoods of node i which is then normalized across all neighboring nodes j using a softmax function as follows

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})} \tag{2.9}$$

In this study, our attention mechanism a will be a single-layer feedforward neural network parametrized by a weight vector $\overrightarrow{a_l} \in \mathbb{R}^{F'}$ and $\overrightarrow{a_r} \in \mathbb{R}^{F'}$, followed by a LeakyReLU nonlinearity (with negative input slope 0.2). Let $\cdot^T$ represent transposition and $||$ represent concatenation. The coefficients computed by our attention mechanism may be expressed as:

$$\alpha_{ij} = \frac{\exp\Big(\text{LeakyReLU}\Big(\overrightarrow{a_l}^T \mathbf{W_l} \overrightarrow{h_i} + \overrightarrow{a_r}^T \mathbf{W_r} \overrightarrow{h_j}\Big)\Big)}{\sum_{k \in \mathcal{N}_i} \exp\Big(\text{LeakyReLU}\Big(\overrightarrow{a_l}^T \mathbf{W_l} \overrightarrow{h_i} + \overrightarrow{a_r}^T \mathbf{W_r} \overrightarrow{h_k}\Big)\Big)} \tag{2.10}$$

Once the normalized attention coefficients are determined, it is fed into the GAT formulation in Equation 2.7 to compute the final output feature representation after linear

combination **W** and nonlinear (σ) operation. It is important to note that GAT is agnostic to the choice of attention mechanism, a. In this study, the attention mechanism is a simple single-layer neural network, but it can be replaced with any other complex learning model.

2.3.2 Multi-Headed Attention

Although a graph attention layer is well-behaved, since the parameters of the attention mechanism are jointly trained, it can sometimes lead to convergence issues. To avert the convergence issues and stabilize the learning process of the attention mechanism, multi-headed attention can be used, as proved effective by Vaswani et al. (2017). Figure 2.4 the

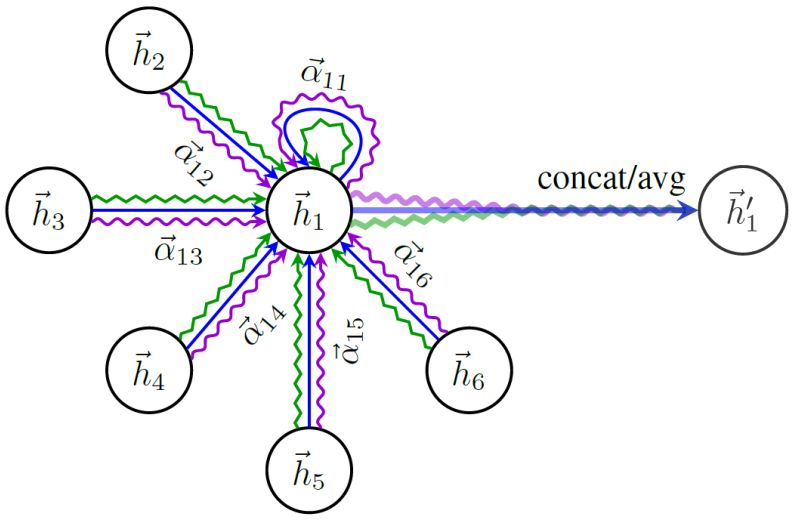


Figure 2.4: An illustration of multi- head attention (with K = 3 heads) by node 1 on its neighborhood. Different arrow styles and colors denote independent attention computations. The aggregated features from each head are concatenated or averaged to obtain $\overrightarrow{h_i}'$. Source: Veličković et al. (2017).

schematics of graph attention layer with $K = 3$ attention heads. The target node in the figure is h_1 with five neighboring nodes. The three different colors represent three attention heads where attention coefficients are computed individually which are then concatenated

or averaged to compute the output features.

Again, following the similar formulations as highlighted in Equation 2.7 and Equation 2.5, the multi-headed attention is mathematically represented as

$$\mathbf{h}_v^{(l)} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{u\in N(v)}\alpha_{vu}\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}\right) \tag{2.11}$$

where K is the number of attention heads. GATs are computationally efficient as the computation of attentional coefficients can be parallelized across all edges of the graph in addition to parallelizing aggregation across all nodes. Since the main idea of the attention mechanism is to only focus on important parts of the input graph, embedding is highly localized thus capturing local features. Finally, the attention weights can transferred from one part of the graph to another so it does not depend on the global graph structure and thus make it an inductive learning process.

2.4 Modified GAT (GATv2)

Although GAT is a state-of-the-art graph-based architecture, Brody et al. (2021) showed, through a controlled implementation, that GAT might have a limited ability to capture attention because it relies on a static attention mechanism. To address this issue, they proposed a simple modification by altering the order of operations. They simply apply "a" layer after the non-linearity and the "W" layer after the concatenation, effectively applying an MLP to compute the score for each query-key pair. This, they argued, has the ability to enhance the attentiveness and expressiveness of GAT dynamically. The modified GAT is hereafter referred to as GATv2.

The order of operation in the conventional GAT is given by

$$e(h_i, h_j) = LeakyReLU(a^T[Wh_i||Wh_j]) \tag{2.12}$$

Algorithm 2 Graph Attention Network (GAT)

Input: Graph $G = (V, E)$, Node features X, Number of attention heads K

Initialize learnable parameters: $\mathbf{W}^{\text{in}}$, $\mathbf{W}^{\text{out}}$, $\mathbf{a}$

function MultiHeadAttention(X, A)

Initialize empty list of output features: outputs = [] **for** k *in range(K)* **do**

Compute attention coefficients:

$$\alpha_{ij}^{(k)} = \text{softmax}\left(\text{LeakyReLU}\left(\mathbf{a}^{(k)} \cdot [\mathbf{W}^{\text{in}} X_i, \mathbf{W}^{\text{in}} X_j]\right)\right)$$

Compute weighted sum of neighboring features:

$$h_i^{(k)} = \sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(k)} \cdot \mathbf{W}^{\text{out}} X_j$$

Append $h_i^{(k)}$ to outputs

Concatenate and normalize outputs: H = concatenate(outputs), H = LayerNorm(H)

Return: Output features H

end function

Compute adjacency matrix: $A = \text{preprocess}(A)$

Apply self-attention: $H = \text{MultiHeadAttention}(X, A)$

Perform graph-level pooling if needed

Make predictions or use for downstream tasks

whereas the order of operation in the modified GAT is as follows

$$e(h_i, h_j) = a^T LeakyReLU(W\dot{[}h_i||h_j]) \tag{2.13}$$

CHAPTER 3

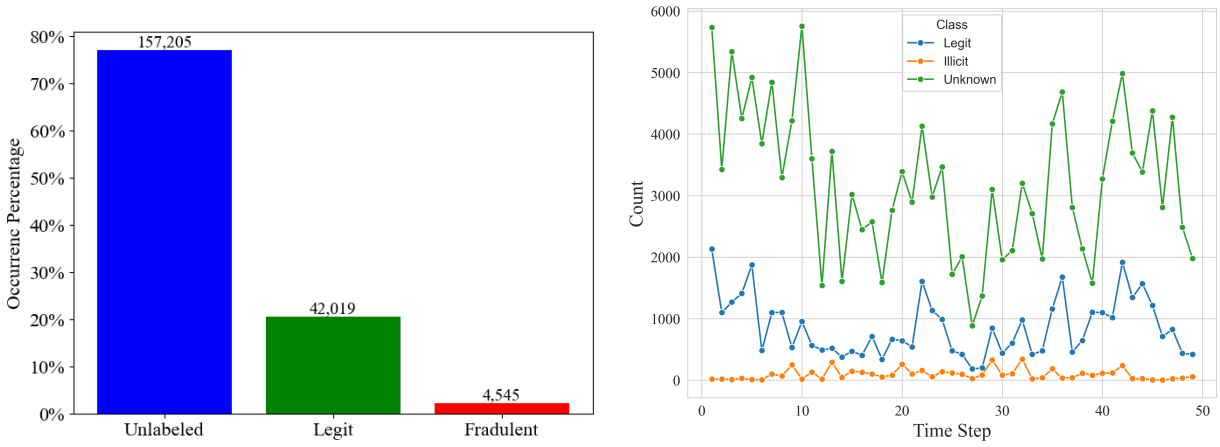
Fraud Detection Using GNNs

The various graph-based neural network models detailed in Chapter 2 are implemented in this chapter. In particular, GCN, GAT, and modified GAT (namely GATv2) are implemented to detect fraudulent Bitcoin transactions. The performance of the three models is compared in detail. The objective of implementing GCN is to use it as a benchmark to compare how self-aggregation and attention mechanisms in GAT enhance the ability of GNN in learning graph representation.

3.1 Dataset

The dataset used in this study is an anonymized transaction graph collected from the Bitcoin blockchain and published by Elliptic (Weber et al., 2019). It is one of the world's largest labeled transaction datasets publicly available in any cryptocurrency. The transaction graph is made of 203,769 nodes and 234,355 edges. A node in the graph represents a transaction while an edge can be viewed as the flow of bitcoin between two transactions. Each node/transaction has 166 features, which have been anonymized due to privacy and intellectual property reasons. The nodes are labeled into three classes, namely, licit, illicit, or unknown. Figure 3.1a breakdown the dataset by class. The vast majority of data (more than 75%) is unlabeled, more than 20% of data is labeled as legit, and about 2.2% of data is labeled as fraudulent. Figure 3.1b presents the number of transactions per time step. The time step of each node represents the timestamp a transaction was tallied in the Bitcoin network. There are 49 evenly spaced time steps with each representing two weeks. Each

time step contains a single connected component of transactions and there are no edges connecting the different time steps.



(a) (b)

Figure 3.1: a) Number of transactional nodes per class; b) loss ratio curves of 609 buildings as a function of the intensity measures.

Out of the 166 anonymized futures, the first 94 represent local information about the transaction such as time of the transaction, transaction fee, output volume, number of incoming or outgoing Bitcoins, etc. The final 72 features are engineered features that include aggregated information about transactions of one-hop neighbors within a time step. Figure 3.2 illustrates the transaction graph for a time step (18^{th} time step to be precise), and transactions that included a fraudulent transaction.

3.2 Model Training

The GNN models are implemented using the *PyTorch Geometric* package in Python (Fey and Lenssen, 2019). In total, three GNN models are built using the elliptic dataset. The following sections discuss in detail the architecture of the implemented models and their performance evaluations.

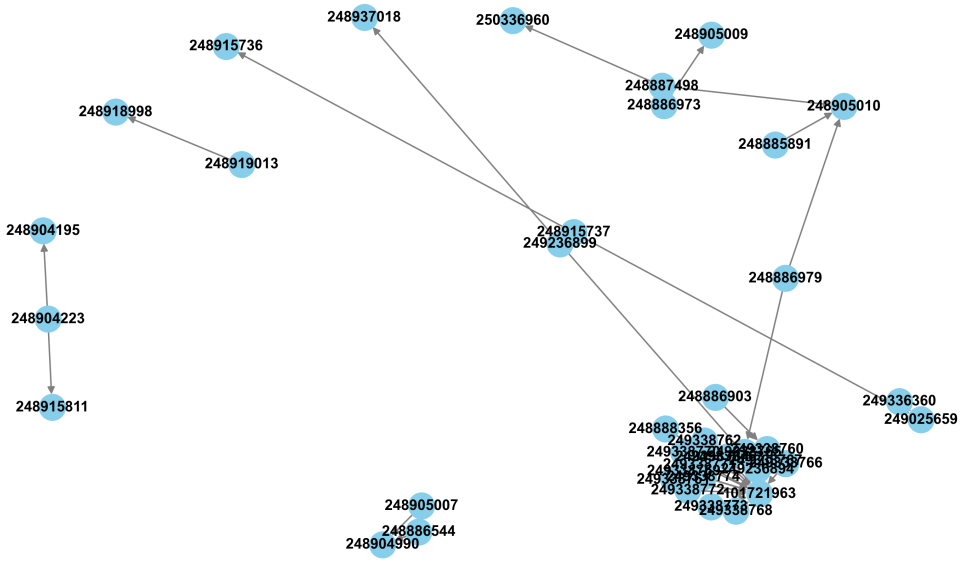


Figure 3.2: Graphical visualization of Bitcoin transactions at time step 18.

3.2.1 GCN

The first implementation is the classic graph convolution network. Figure 3.3 presents the architecture encoded. It consists of three convolutional layers followed by a linear classifier. The three convolutional layers decrease output feature dimensions from 165 to 128 to 64 respectively and the linear classifier maps the final embedding space to a single dimension while considering bias. In each forward pass, a layer applies a graph convolution operation to update node representations based on their local neighborhood information in the graph. After each convolutional layer, nonlinearity is introduced by applying a hyperbolic tangent (i.e., tanh) activation function. The output of the last graph convolutional layer (namely *conv3*) represents the embeddings of nodes in a lower-dimensional space as the output dimension is two. The tanh activation function is applied again to constrain the embeddings within a bounded range. Finally, the embeddings are passed through the linear classifier to predict the target variable. A sigmoid activation function is applied to ensure that the

output falls within the range [0, 1].

```
GCN(
  (conv1): GCNConv(165, 128)
  (conv2): GCNConv(128, 64)
  (conv3): GCNConv(64, 2)
  (classifier): Linear(in_features=2, out_features=1, bias=True)
)
```

Figure 3.3: Model architecture of classical GCN with three convolutional layers and a linear classifier.

3.2.2 GAT

In this study, two versions of GAT are implemented. The first iteration of GAT implementation directly adopts the architecture proposed by Veličković et al. (2017). As previously discussed in Section 2.3.1, the attention coefficient is computed as

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{a_l}^T \mathbf{W_l} \overrightarrow{h_i} + \overrightarrow{a_r}^T \mathbf{W_r} \overrightarrow{h_j}\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\overrightarrow{a_l}^T \mathbf{W_l} \overrightarrow{h_i} + \overrightarrow{a_r}^T \mathbf{W_r} \overrightarrow{h_k}\right)\right)} \tag{3.1}$$

As depicted in Figure 3.4, two message-passing layers are included with the purpose of applying the attention mechanism to aggregate information from neighboring nodes and produce updated node features. The updated node features are supplied to a sequential multi-layer perceptron (MLP). The MLP consists of linear layers followed by dropout and ReLU activation. The output of the MLP is passed through a sigmoid activation function to produce the final output of the model, which includes the probabilities of the predicted classes.

3.2.3 GATv2

In one of the important contributions to GAT, Brody et al. (2021) showed that GAT has a limited ability to capture attention because it relies on a static attention mechanism. The

```
GAT(
  (conv1): GATConv(165, 128, heads=2)
  (conv2): GATConv(256, 128, heads=2)
  (post_mp): Sequential(
    (0): Linear(in_features=256, out_features=128, bias=True)
    (1): Dropout(p=0.5, inplace=False)
    (2): Linear(in_features=128, out_features=1, bias=True)
  )
)
```

Figure 3.4: Model architecture of GAT with two convolutional layers and an MLP classifier.

authors showed, through a controlled problem, that by modifying the order of operations one can dynamically enhance the attentiveness and expressiveness of GAT. The modification is shown in Equation 3.2. Other than this minor modification, the architecture and implementation of GAT and GATv2 are exactly the same as described in Figure 3.5

$$\alpha_{ij} = \frac{\exp\left(\overrightarrow{a}^T \text{LeakyReLU}\left(\mathbf{W_l}\overrightarrow{h_i} + \mathbf{W_r}\overrightarrow{h_j}\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\overrightarrow{a}^T \text{LeakyReLU}\left(\mathbf{W_l}\overrightarrow{h_i} + \mathbf{W_r}\overrightarrow{h_k}\right)\right)} \tag{3.2}$$

```
GATv2(
  (conv1): GATv2Conv(165, 128, heads=2)
  (conv2): GATv2Conv(256, 128, heads=2)
  (post_mp): Sequential(
    (0): Linear(in_features=256, out_features=128, bias=True)
    (1): Dropout(p=0.5, inplace=False)
    (2): Linear(in_features=128, out_features=1, bias=True)
  )
)
```

Figure 3.5: Model architecture of modified GAT with two convolutional layers (with modified attention mechanism) and an MLP classifier.

Figure 3.6 presents the training loss for the three models. As expected, the training loss for GCN starts slightly higher and remains noticeably higher as the number of epochs increases. On the contrary, the loss for GAT and GATv2 sees a drastic drop after 5 epochs

after which the loss decreases gradually. The rate and margin of reduction for GAT and GATv2 are identical to GATv2 with a slight and seemingly insignificant lead. This highlights the enhanced performance of GAT models while giving an early indication that the dynamic representation in GATv2 has an insignificant impact on the performance of the attention mechanism.

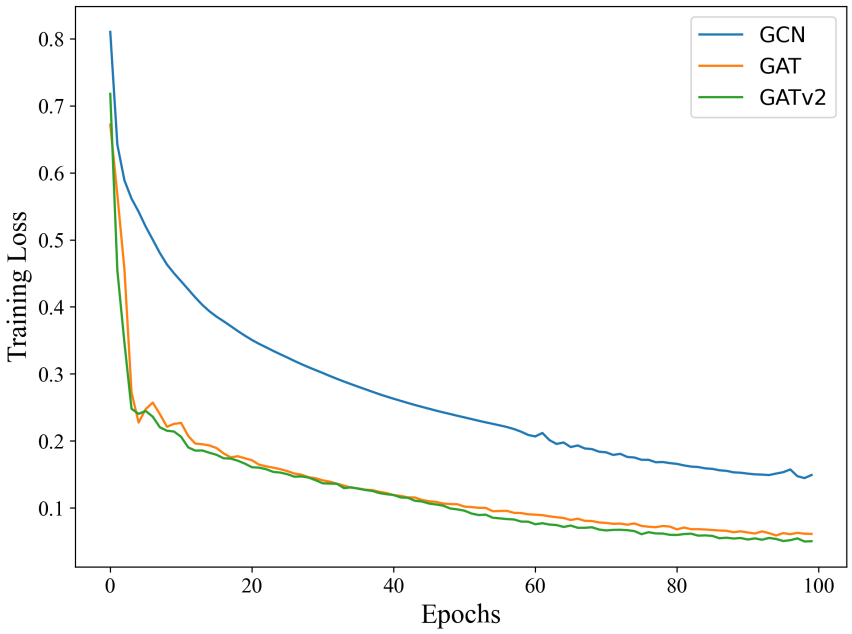


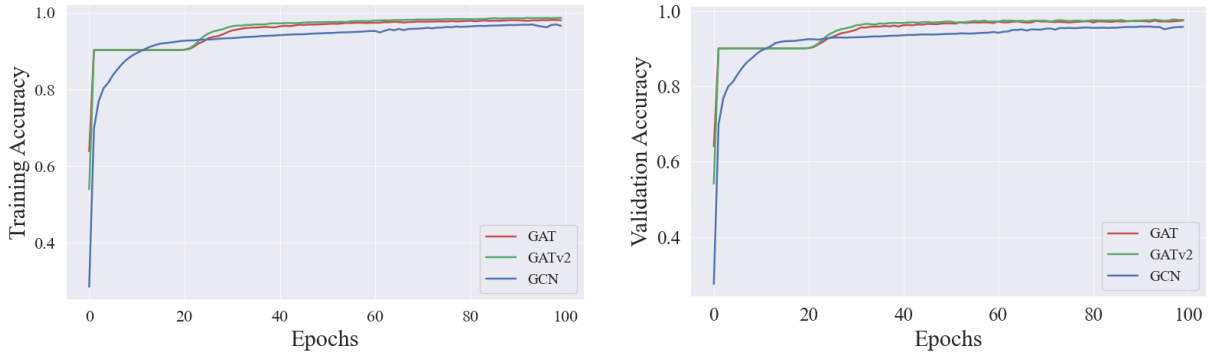
Figure 3.6: Comparison of training losses as a function of epochs for GCN, GAT, and GATv2 models.

3.3 Model Evaluation

To compare the three models, an extensive model evaluation is undertaken based on the best model from the training phase. Figure 3.7 demonstrates the training and validation accuracy. Figure 3.7a shows that all three models start to capture the data well after a few epochs. It is evident that the GAT and GATv2 reach 90% accuracy instantaneously but

GCN requires a few additional epochs. The training accuracy of the GAT model is better than the GCN model. A near-identical trend is observed for the validation with GAT models outperforming the GCN model as shown in Figure 3.7b. The plots for F1-Micro, F1-Macro, Precision, Recall, and AUC-ROC are presented in Appendix A. The accuracy is computed as

$$\text{accuracy} = \frac{TP + TN}{TP + FN + TN + FP} \tag{3.3}$$



(a) (b)

Figure 3.7: a) Training and b) validation accuracy of the three models.

Figure 3.8 displays confusion matrices for the three models. Consistent with previous observations, GAT and GATv2 perform better with higher rates of true negatives, true positives, false positives, and false negatives. For instance, the true negatives of GCN, GAT, and GATv2 are 8260, 8318, and 8306, respectively. It is evident that the misclassification (false positives and false negatives) is lower in GAT-based models.

3.4 A Note on Explainability

To explain the model and its performance, we plot the transaction graph at time step 28 as shown in Figure 3.9. The blue, green, and red colors represent unlabeled, legit, and fraudulent, respectively. A graph like Figure 3.9 can be instrumental in explaining the

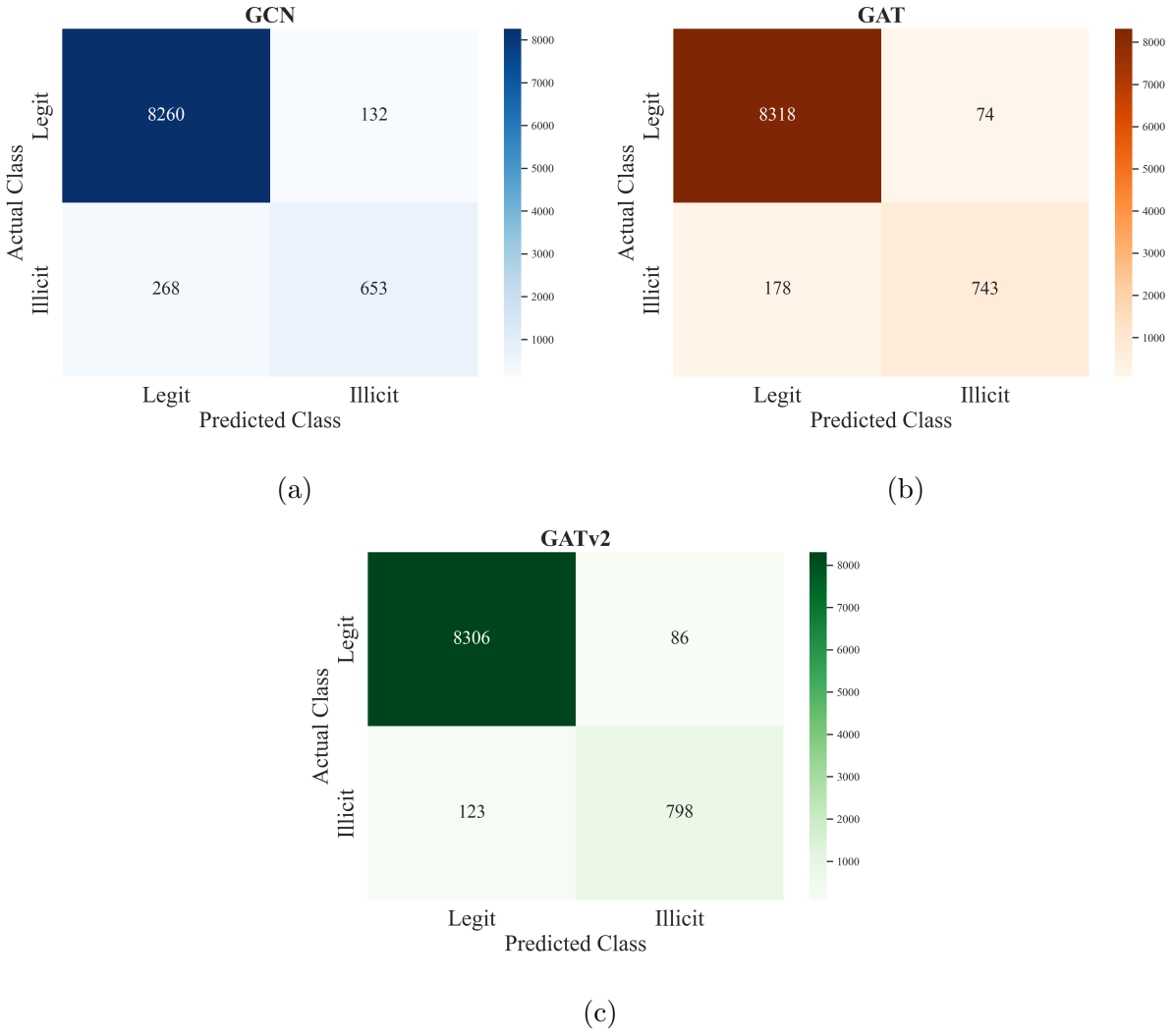


Figure 3.8: Confusion matrix of a) GCN; b) GAT; c) GATv2.

model. However, since the data used in this study is anonymized, the trend in Figure 3.9 is hardly explainable.

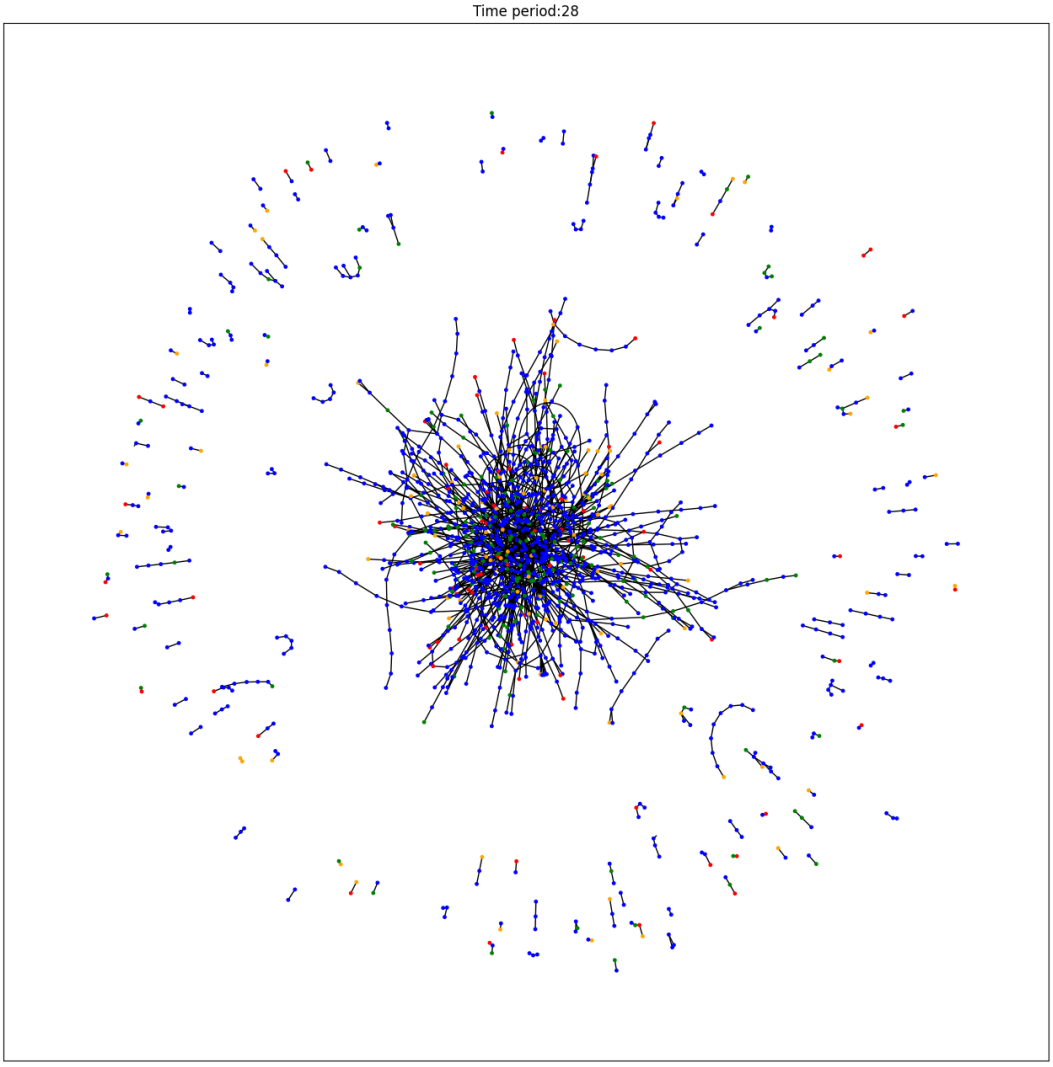


Figure 3.9: Transaction graph with predicted label using GAT for time step 28.

CHAPTER 4

Conclusions

4.1 Summary and Findings

Graph neural networks (GNN) are gaining popularity for various prediction tasks involving geometric data such as financial transactions. In this study, we investigated the effectiveness of various GNN models such as graph convolutional network (GCN), graph attention network (GAT), and modified graph attention network (GATv2). First, an extensive background in graph-based machine learning is presented along with different types of learning and node embeddings. In particular, the study focused on GAT and attention mechanism as a potential alternative to traditionally used GCN models. The use of masked self-attentional layers in GAT is discussed to address the shortcomings of conventional GCNs. One of the advantages of the GAT architecture is that it is computationally efficient and does not require spectral features of the entire graph upfront. In the second half, the theoretical background on GCN, GAT, and GATv2 are described in detail. Finally, the three GNN models are implemented to predict fraudulent Bitcoin transactions. The extensive model evaluation indicated that GAT performs better but the performance between GAT and GATv2 are comparable.

4.2 Limitations and Future Studies

While GATs exhibited superior performance compared to GCN, there are several limitations that could lead to potential future studies. First, the dataset could be expanded to include more Bitcoin transactions. The majority of the transactions were not labeled, which is to be

expected because the majority of financial transactions are non-fraudulent. However, having a dataset that has more labeled data could be beneficial in developing the models. In this study, the GNN models were implemented only on Bitcoin transactions. Future studies could extend the implementation to other financial transaction datasets. Moreover, the current implementation included time steps but the built model was static. In practical application, having a dynamic model that updates as more transaction data become available in real-time could be one potential way to deal with the changing dynamics of fraud and camouflage. Furthermore, the data used in the current study had anonymized features which hindered our ability to develop an explainable model. Future studies should focus on the interpretability of graph-based deep learning models to understand the underlying fraud detection mechanisms.

APPENDIX A

Appendix

The following abbreviations are used in this appendix

1. TP = True Positive
2. FP = False Positive
3. FN = False Negative
4. TN = True Negative

A.1 Precision

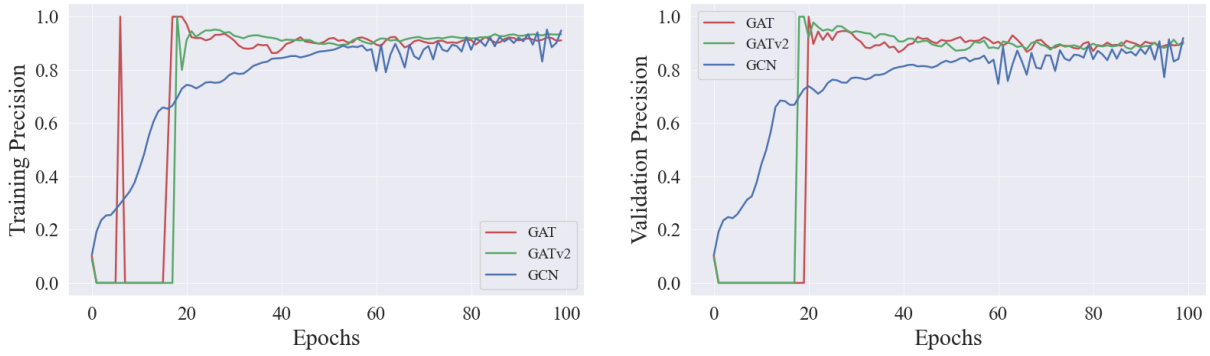
Precision measures a model's ability to avoid false positives. It is given by,

$$\text{Precision} = \frac{TP}{TP + FN} \tag{A.1}$$

A.2 Recall

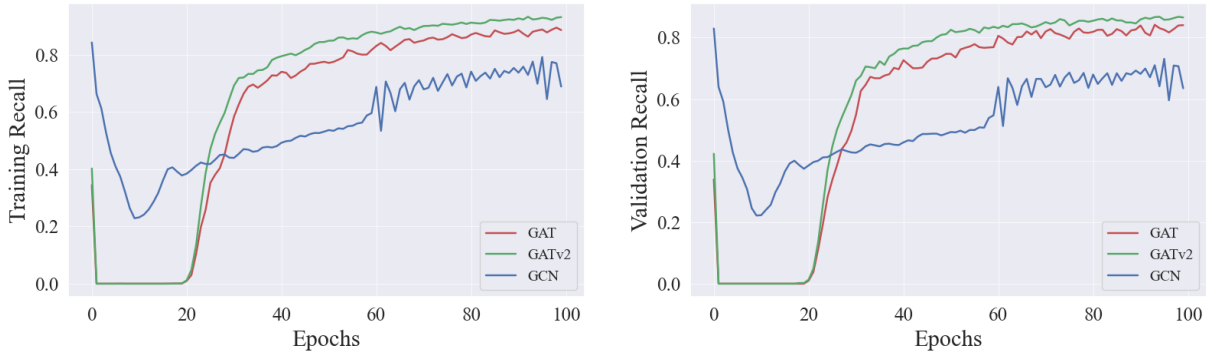
Recall, also known as sensitivity, measures a model's ability to capture all positive instances. It is computed as

$$\text{Precision} = \frac{TP}{TP + FN} \tag{A.2}$$



(a) (b)

Figure A.1: Precision of GCN, GAT, and GATv2 for a) training; b) validation as a function of the epochs.



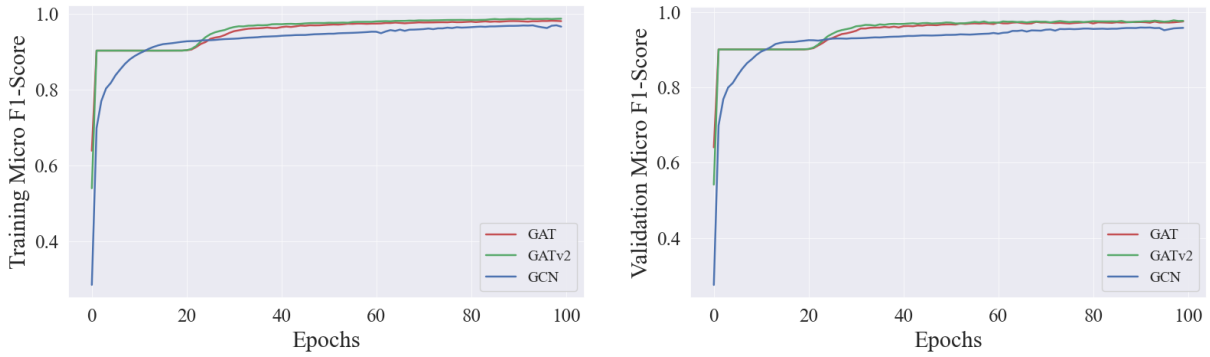
(a) (b)

Figure A.2: Recall of GCN, GAT, and GATv2 for a) training; b) validation as a function of the epochs.

A.3 F1-Micro

The F1 score is the harmonic mean of precision and recall. It provides a single metric to balance both precision and recall.

$$\text{F1 Micro} = \frac{2 \times Precision \times Recall}{Precision \times Recall} \tag{A.3}$$



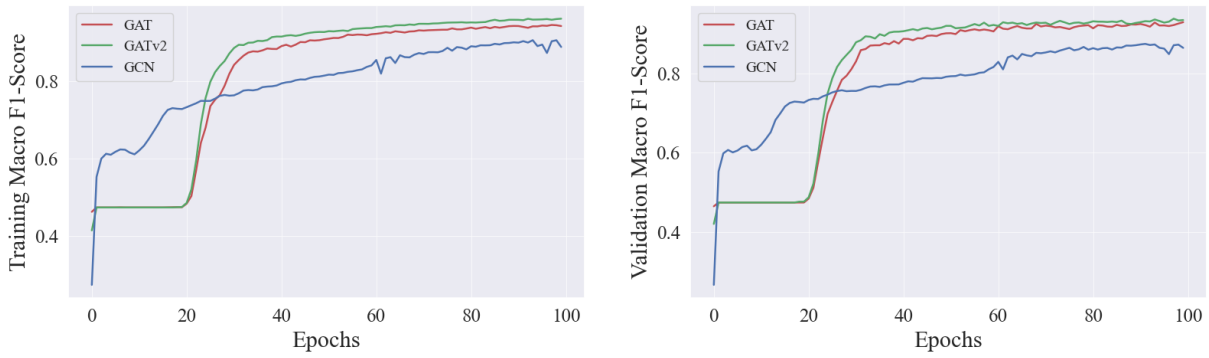
(a) (b)

Figure A.3: F1-Micro of GCN, GAT, and GATv2 for a) training; b) validation as a function of the epochs.

A.4 F1-Macro

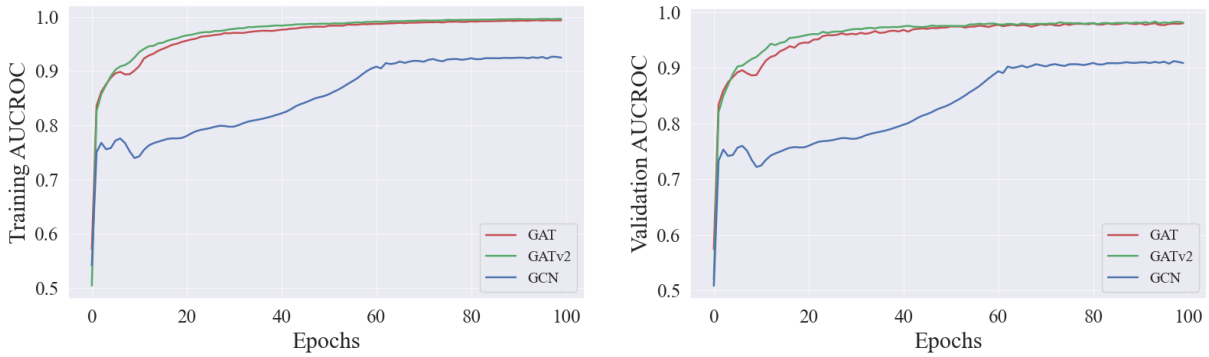
A.5 AUC-ROC

AUC-ROC measures the area under the curve of the receiver operating characteristic (ROC) curve, which plots the true positive rate (recall) against the false positive rate (1 - specificity) for different threshold values. It quantifies the model's ability to distinguish between positive and negative instances across all possible threshold values. It ranges from 0 to 1, where a value closer to 1 indicates better model performance.



(a) (b)

Figure A.4: F1-Macro of GCN, GAT, and GATv2 for a) training; b) validation as a function of the epochs.



(a) (b)

Figure A.5: AUC-ROC of GCN, GAT, and GATv2 for a) training; b) validation as a function of the epochs.

Bibliography

Adamic, L.A., Adar, E., 2003. Friends and neighbors on the web. Social networks 25, 211–230.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473 .

Borgwardt, K.M., Kriegel, H.P., 2005. Shortest-path kernels on graphs, in: Fifth IEEE international conference on data mining (ICDM'05), IEEE. pp. 8–pp.

Brody, S., Alon, U., Yahav, E., 2021. How attentive are graph attention networks? arXiv preprint arXiv:2105.14491 .

Dou, Y., Liu, Z., Sun, L., Deng, Y., Peng, H., Yu, P.S., 2020. Enhancing graph neural network-based fraud detectors against camouflaged fraudsters, in: Proceedings of the 29th ACM international conference on information & knowledge management, pp. 315–324.

Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., Yin, D., 2019. Graph neural networks for social recommendation, in: The world wide web conference, pp. 417–426.

Fey, M., Lenssen, J.E., 2019. Fast graph representation learning with PyTorch Geometric, in: ICLR Workshop on Representation Learning on Graphs and Manifolds.

Frasconi, P., Gori, M., Sperduti, A., 1998. A general framework for adaptive processing of data structures. IEEE transactions on Neural Networks 9, 768–786.

Grover, A., Leskovec, J., 2016. node2vec: Scalable feature learning for networks, in: Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 855–864.

Hamilton, W., Ying, Z., Leskovec, J., 2017a. Inductive representation learning on large graphs. Advances in neural information processing systems 30.

Hamilton, W.L., Ying, R., Leskovec, J., 2017b. Representation learning on graphs: Methods and applications. arXiv preprint arXiv:1709.05584 .

Hammond, D.K., Vandergheynst, P., Gribonval, R., 2011. Wavelets on graphs via spectral graph theory. Applied and Computational Harmonic Analysis 30, 129–150.

Joachims, T., 1998. Text categorization with support vector machines: Learning with many relevant features, in: European conference on machine learning, Springer. pp. 137–142.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., et al., 2021. Highly accurate protein structure prediction with alphafold. Nature 596, 583–589.

Katz, L., 1953. A new status index derived from sociometric analysis. Psychometrika 18, 39–43.

Kipf, T.N., Welling, M., 2016. Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 .

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. nature 521, 436–444.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE 86, 2278–2324.

Lu, Y.J., Li, C.T., 2020. Gcan: Graph-aware co-attention networks for explainable fake news detection on social media, in: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pp. 505–514.

McCallum, A., Nigam, K., et al., 1998. A comparison of event models for naive bayes text classification, in: AAAI-98 workshop on learning for text categorization, Madison, WI. pp. 41–48.

Micheli, A., 2009. Neural network for graphs: A contextual constructive approach. IEEE Transactions on Neural Networks 20, 498–511.

Monti, F., Frasca, F., Eynard, D., Mannion, D., Bronstein, M.M., 2019. Fake news detection on social media using geometric deep learning. arXiv preprint arXiv:1902.06673 .

Niwattanakul, S., Singthongchai, J., Naenudorn, E., Wanapu, S., 2013. Using of jaccard coefficient for keywords similarity, in: Proceedings of the international multiconference of engineers and computer scientists, pp. 380–384.

Perozzi, B., Al-Rfou, R., Skiena, S., 2014. Deepwalk: Online learning of social representations, in: Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 701–710.

Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G., 2008. The graph neural network model. IEEE transactions on neural networks 20, 61–80.

Shervashidze, N., Schweitzer, P., Van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M., 2011. Weisfeiler-lehman graph kernels. Journal of Machine Learning Research 12.

Sperduti, A., Starita, A., 1997. Supervised neural networks for the classification of structures. IEEE Transactions on Neural Networks 8, 714–735.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. Advances in neural information processing systems 30.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y., 2017. Graph attention networks. arXiv preprint arXiv:1710.10903 .

Vishwanathan, S.V.N., Schraudolph, N.N., Kondor, R., Borgwardt, K.M., 2010. Graph kernels. Journal of Machine Learning Research 11, 1201–1242.

Wang, D., Lin, J., Cui, P., Jia, Q., Wang, Z., Fang, Y., Yu, Q., Zhou, J., Yang, S., Qi, Y., 2019. A semi-supervised graph attentive network for financial fraud detection, in: 2019 IEEE International Conference on Data Mining (ICDM), IEEE. pp. 598–607.

Wang, D., Zhang, Z., Zhou, J., Cui, P., Fang, J., Jia, Q., Fang, Y., Qi, Y., 2021. Temporal-aware graph neural network for credit risk prediction, in: Proceedings of the 2021 SIAM International Conference on Data Mining (SDM), SIAM. pp. 702–710.

Weber, M., Domeniconi, G., Chen, J., Weidele, D.K.I., Bellei, C., Robinson, T., Leiserson, C.E., 2019. Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics. arXiv preprint arXiv:1908.02591 .

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Philip, S.Y., 2020. A comprehensive survey on graph neural networks. IEEE transactions on neural networks and learning systems 32, 4–24.

Xiang, S., Zhu, M., Cheng, D., Li, E., Zhao, R., Ouyang, Y., Chen, L., Zheng, Y., 2023. Semi-supervised credit card fraud detection via attribute-driven graph representation, in: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 14557–14565.

Zhang, S., Yin, H., Chen, T., Hung, Q.V.N., Huang, Z., Cui, L., 2020. Gcn-based user representation learning for unifying robust recommendation and fraudster detection, in: Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval, pp. 689–698.

Zhong, Q., Liu, Y., Ao, X., Hu, B., Feng, J., Tang, J., He, Q., 2020. Financial defaulter detection on online credit payment via multi-view attributed heterogeneous information network, in: Proceedings of The Web Conference 2020, pp. 785–795.

Zhou, F., Yang, Q., Zhong, T., Chen, D., Zhang, N., 2020a. Variational graph neural networks for road traffic prediction in intelligent transportation systems. IEEE Transactions on Industrial Informatics 17, 2802–2812.

Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., Sun, M., 2020b. Graph neural networks: A review of methods and applications. AI open 1, 57–81.