Graph Neural Networks Unveiled: A Novel Taxonomy and Comparative Analysis

NS Surya, Ramnaresh Ulaganathan, M Anbazhagan

March 2024

Abstract

As computational power experiences an exponential surge, the realm of deep learning and intricate neural network structures has become a tangible and powerful force. The prowess of deep learning surpasses even the most sophisticated machine learning algorithms, showcasing its capability to address intricate challenges. In recent years, substantial research has been conducted into the integration of graph databases into major corporate systems, with the goal of replacing costly relational database management systems (RDBMS). The cost of RDBMS increases as the number of tables and records grows, and as previously stated, where there is data, there is a demand for data analysis and forecasts. This necessity has given rise to Graph Machine Learning (Graph ML) and Graph Neural Networks, which solve the constraints of current machine learning and deep learning techniques. This study will walk readers through the transition from classic neural network structures to graph neural network topologies, providing insights into the most recent advances in this field. We will consider a range of benchmark datasets, including Cora, Facebook, Proteins, AIFB, PubMed, Wikipedia, Cite-Seer, etc. Our analysis will involve the application of diverse models for tasks such as node-level, edge-level, or graph-level predictions.

1 Introduction

The preference for graphs over tables has significantly increased in the last few years. According to a Gartner report, "The application of graph processing and graph databases will

grow at 100% annually through 2022 to continuously accelerate data preparation and enable more complex and adaptive data science."[1] With the proliferation of complex relationships and dependencies in various domains, graphs offer a natural and intuitive framework for modelling and analyzing data. From social networks to recommendation systems, from biological networks to transportation networks, graphs find applications across diverse domains, facilitating efficient storage, retrieval, and analysis of interconnected data. Their ability to capture both the structure and semantics of relationships enable sophisticated data querying, pattern discovery, and predictive analysis, making graphs indispensable in modern data management and analysis tasks.

In real-life applications, graphs play a pivotal role across a spectrum of products offered by leading tech companies. For example, Google's search engine uses Semantic Search, which leverages Knowledge Graphs to improve search results with contextual knowledge[2]. Facebook uses social networks to suggest friends and tailor content, increasing user engagement[3]. Furthermore, Amazon uses recommendation graphs to recommend goods based on consumer browsing and purchase activity, which improves the shopping experience[4]. Knowledge graphs (KGs) outperform typical tabular data for machine learning (ML) tasks, providing a richer and more expressive representation of interrelated knowledge. Unlike tabular data, KGs capture intricate relationships between entities, attributes, and their semantic meanings. This comprehensive picture helps ML models to detect complex patterns and connections that would otherwise go unnoticed in flat datasets. Graphs are used in ML tasks such as question-answering, storing research, supply chain management, etc[5]. Furthermore, KGs enable the smooth integration of heterogeneous data sources, resulting in a unified framework for using diverse data. This improved representation promotes more robust and complete AI/ML solutions, allowing applications to extract insights from linked data across domains more effectively and precisely.

The transition from Relational Database Management Systems (RDBMS) to graph databases is being driven by a paradigm shift, particularly in the era of expanding cloud computing capabilities. Graph databases, such as Neo4j, OrientDB, and AWS Neptune, outperform RDBMS in cases involving big datasets. Their unstructured representation, which uses nodes

and edges, allows for dynamic flexibility and better visualization of complex relationships—a significant benefit over RDBMS's structured, table-based approach. The inherent ability to efficiently traverse relationships promotes graph databases as a tempting solution, stressing agility and adaptability in modern data management architectures.[6]

The shift from RDBMS to graph databases has effectively addressed inherent limitations, giving rise to specialized Machine Learning (ML) algorithms tailored for graph-related challenges. These algorithms have proven invaluable for companies and researchers in predicting intricate details from graphs. Graph Machine Learning is a subset of machine learning that uses graph data to perform predictive and prescriptive tasks. The most common tasks in supervised learning are node feature prediction, link prediction, graph property predictions. While the common tasks in unsupervised learning are representation learning, community detection, centrality, and pathfinding. While traditional graph algorithms like PageRanker and Dijkstra's Shortest Path may tackle centrality and path finding problems, machine learning algorithms require low-dimensional representations of graph data for node, edge, and graph-level tasks. Several techniques are used to encode properties of nodes, links, or even the entire graph into low-dimensional vectors of either continuous or discrete (binary) values. These techniques rely on matrix factorizations, random walks, random projections, or encoder-decoder models. A couple of examples include node2vec (random walk) and HOPE (factorization).[7][8]

[9]The need for DL in the realm of graphs emerged. However, a challenge arises as conventional DL methods, like Convolutional Neural Networks (CNN) and Artificial Neural Networks (MLP), prove suboptimal in addressing graph-related issues. These methods neglect the crucial information about the interconnectedness of nodes with their neighbors, limiting their performance on graph data. To address this concern, we required a new set of algorithms which led to the creation of graph neural networks. The benefits of Graph Neural Networks (GNNs) go beyond their ability to process graph-structured data. GNNs excel at capturing complicated patterns and dependencies within these systems. Unlike classic neural networks, GNNs use message-passing mechanisms to efficiently propagate information around the graph. This novel approach enables GNNs to examine both inherent attributes

and relationships with nearby nodes, resulting in a more comprehensive understanding of the graph's dynamics. The hierarchical feature aggregation improves pattern discernment at different scales, resulting in a more complete study. Incorporating graph convolutional techniques created specifically for graph topology ensures that the model recognizes patterns driven by both node properties and network connectivity. GNNs can adapt to different graph sizes and effectively absorb structural information, which improves their prediction powers. This multifaceted approach, encompassing message passing and graph-aware operations, positions GNNs as powerful tools for diverse graph-related tasks, outperforming traditional neural networks in capturing the intricacies of relational data.

2 Background of GNNs

This section aims to provide readers with a thorough understanding of the concept of graph neural networks, along with an explanation of how the various architectures operate and some of their applications.

Delving into the rich history of Graph Neural Networks (GNNs), their roots extend to the 1980s, where academic exploration ignited a curiosity about employing neural networks for graph-structured data. The 1990s witnessed a pivotal moment with the advent of recursive neural networks, spurring endeavours to adapt these principles to graphs, giving rise to the nascent concept of graph neural networks. Despite these early strides, the true ascendancy of GNNs materialized in the 2010s, spurred by a confluence of factors and a resurgent interest in leveraging neural structures for graph-centric applications. This evolutionary trajectory reached a watershed moment in 2017 when Thomas N Kipf introduced graph convolutional networks (GCNs). Drawing inspiration from the original Convolutional Neural Network (CNN) architectures, GCNs transcended their predecessors, showcasing superior performance across an array of benchmark datasets. This milestone underscored the iterative refinement and transformative potential of GNNs, solidifying their status as a formidable paradigm in the realm of graph-based learning.

Building on this historical narrative, a diverse tapestry of GNN architectures has since

emerged, each weaving in elements from traditional neural network structures. Among these, Graph Attention Networks (GAT), Graph Sample and Aggregated (GraphSAGE), Graph Isomorphic Network (GIN), Relational Graph Convolutional Network (RGCN), Temporal Graph Networks (TGN), stand out as noteworthy contributions. In the context of this paper's objective to demystify GNNs for a broader audience, a distilled definition is proposed: GNNs can be conceptualized as neural networks augmented with message passing layers, succinctly denoted as GNN = NN + MP. This encapsulation succinctly conveys the essence of GNNs, harmonizing the principles of conventional neural networks with the distinctive trait of message passing—an elemental operation crucial within the context of graph-structured data. This historical odyssey coupled with contemporary architectural diversity paints a comprehensive picture of the GNN landscape, signifying its journey from nascent explorations to a pivotal force in modern machine learning paradigms. Next let us try and understand the various architectures involved in GNNs.

2.1 Multi-layer Perceptron

A multi-layer perceptron (MLP) is a category of artificial neural network (ANN) comprised of a feedforward arrangement of numerous layers of nodes or neurons. Each neuron in a given layer establishes a weighted connection with every neuron in the following layer, without engaging in any cycle formation. A possible question at this time is whether a single hidden layer network can adequately encompass a broader range of models. The essence of the Universal Approximation Theorem is precisely this. A very intriguing property of the theorem is that it is now possible to approximate any continuous function by mapping inputs to outputs via a hidden layer. In practice, empirical evidence suggests that stacking multiple hidden layers is more efficient (regardless of the number of parameters needed) than relying on a single layer to accomplish a given approximation quality. Furthermore, to cover a broader variety of models, a more complex model can be constructed by stacking neurons that are organized in layers as shown in Figure 1.

$$\hat{y} = \varphi_{out}\left(\sum_{i=0}^{6} w_{i0}^{m} h_i^m + b^m\right) \tag{1}$$

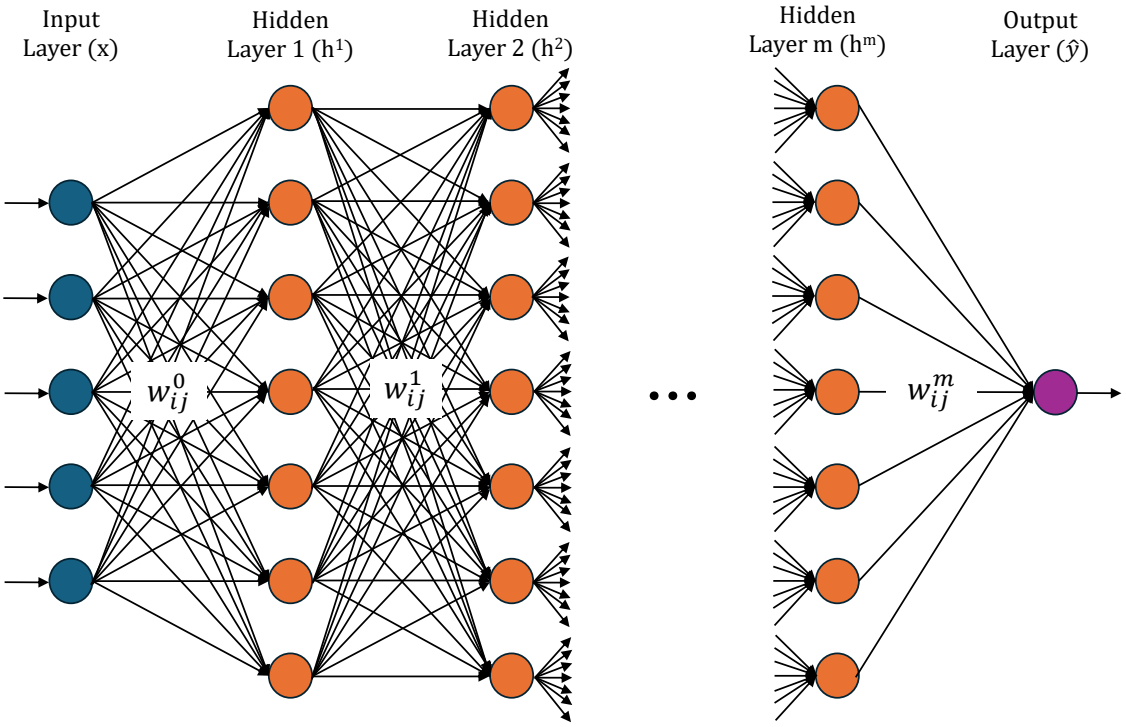

Figure 1: Multi-layer Perceptron with m Hidden layers

$$\forall i, h_i^m = \varphi\left(\sum_{j=0}^{6} w_{ij}^{m-1} h_j^{m-1} + b_i^{m-1}\right) \tag{2}$$

$$\forall i, h_i^2 = \varphi\left(\sum_{j=0}^{6} w_{ij}^1 h_j^1 + b_i^1\right) \tag{3}$$

$$\forall i, h_i^1 = \varphi\left(\sum_{j=0}^{5} w_{ij}^0 x_j + b_i^0\right) \tag{4}$$

The model expressed in equations 1 through 4 is indicative of Figure 1. Certain quantities are fixed by the problem at hand when designing a Multi-Layer Perceptron model for that problem; others are treated as hyper-parameters. The determination of the number of neurons in the input and output layers is contingent upon the dataset's independent variables and the nature of the problem at hand. In the case of binary classification, one neuron is sufficient for the output layer, while n output neurons are required for the n-class classification problem. After these numbers of input and output neurons have been fixed, the

number of hidden neurons and the number of neurons per hidden layer are left as the model's hyper-parameters. An additional crucial hyper-parameter in neural networks is the selection of activation function φ. The output layer possesses an independent activation function φ_{out}, as represented by Equation 1. This is because the activation functions selected for the output layer of a neural network are somewhat problem-dependent.

Although frequently employed with tabular or sequential data, MLPs can also be applied to graphs. However, MLPs make an independent evaluation of each input feature, disregarding the graph's relational structure. This may result in difficulties capturing crucial information regarding the topology of the graph or the neighborhoods of nodes, both of which are essential for numerous graph-related tasks. Furthermore, While graphs can contain an infinite number of nodes and edges, MLPs demand input vectors of a fixed size. Converting variable-size graphs to fixed-size inputs requires pre-processing processes such as graph pooling or padding, which may result in inefficiency or information loss. Also, MLPs that have been trained on a particular graph may exhibit limited generalizability when applied to graphs with distinct structures that they have not yet encountered. MLPs learn fixed transformations that are not dependent on the structure of the graph, which results in this generalization deficiency.

2.2 Vanilla Graph Neural Networks

The notion of extending the convolution operation, which is widely recognized in the domain of images, to graph-structured data gave rise to the GNN concept. Vanilla/Plain GNN represents the most fundamental GNN implementation. A message-passing layer is all that is present to attempt to fill in the gaps left by the MLP. The message-passing layer is nothing but a function with learnable parameters, that is shared by all nodes to enable nodes to update their hidden states. This function known as the local transition function (or message passing function) requires both the current hidden state of a node and its immediate neighbors in order to update its hidden state as given in Equation 5. In Equation 5, $h_i^{(k+1)}$ and h_i^k are the new hidden state and the actual hidden state of node v_i, respectively. $h_{N_i}^k$ is the actual hidden state of v_i's neighbors.

$$h_i^{(k+1)} = \psi\left(h_i^k, h_{N_i}^k\right) \tag{5}$$

On the contrary, the global transition function, denoted as equation 6, is a function with learnable parameters that updates the hidden states of each node in a graph simultaneously. The global transition function necessitates the adjacency matrix of the graph (A) and all current hidden states ($H^{(k)}$) to achieve this. Within the domain of neural network jargon, the global transition function may be referred to as a GNN layer. By iteratively applying the global transition function, the Banach fixed point theorem guarantees that each hidden state will converge to a final value. The objective would be to define node embeddings using these stable hidden states. However, in practice, we will approximate these stable hidden states through the application of the global transition function for a number of iterations (K) that we have predetermined. Each node embedding would contain information within a K-hops neighborhood if the function were to be applied K times as shown in Equation 7.

$$H^{(k+1)} = \psi\left(H^{(k)}, A\right) = \begin{bmatrix} \psi_w\left(h_1^{(k)}, h_{N_1}^{(k)}\right)^T \\ \vdots \\ \psi_w\left(h_{|v|}^{(k)}, h_{N_{|v|}}^{(k)}\right)^T \end{bmatrix} \tag{6}$$

$$H^* \approx \psi_w \circ \psi_w \circ \cdots \circ \psi_w(X, A) \tag{7}$$

A VGNN is composed of several layers, with each layer being tasked with the aggregation and updating of information originating from neighboring nodes. The fundamental concept underlying GNNs is the "message-passing" paradigm, in which during the training process, information is exchanged between nodes. Each node aggregates information from its neighboring nodes during the message-passing phase at each layer. This information is subsequently converted into an informative message. Typically, the message comprises data derived from both the node's internal characteristics and those of its neighboring nodes As depicted in Figure 2. The received messages are utilized by the node in the update phase of each layer to modify its internal representation or embedding. This process allows nodes to integrate data from their immediate vicinity and enhance their depiction. Through the

iterative repetition of the two phases mentioned above, the GNN facilitates the propagation of information throughout the entire graph, thereby enabling nodes to collectively learn and refine their embeddings.

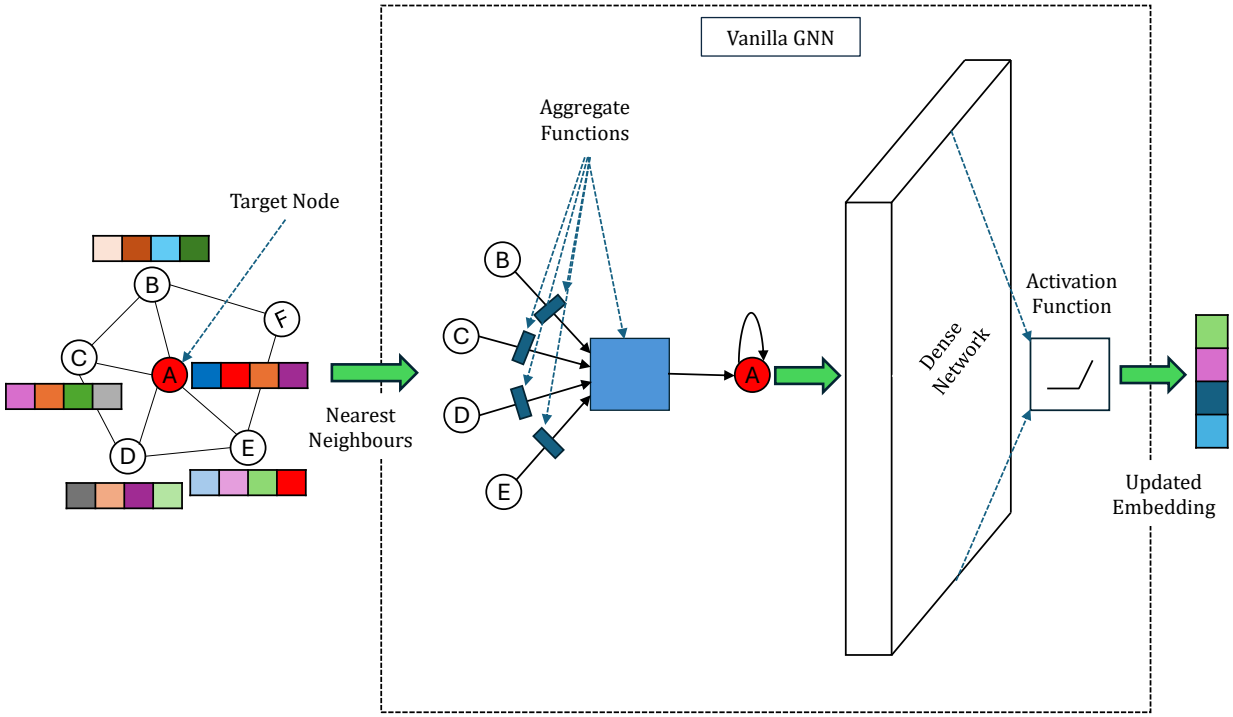

Figure 2: Vanilla Graph Neural Network

2.3 Graph Convolution Network

Convolutional Neural Networks (CNNs) have shown to be extremely effective in extracting complicated features, and they have become an essential component of many Deep Learning models. They've been effective with any dimensions of data. It is their capacity to learn a series of filters and extract more complex patterns. But the images and videos are in Euclidean space; how can we apply this concept to irregular structures such as graphs in non-Euclidean space? Here come the Convolution Theorem and Fourier transform to rescue. The Convolution Theorem states that in the frequency domain, the convolution of two signals (functions) is the component-wise product-wise product of their Fourier transforms. If x and y are the signals and F represent the Fourier transformation function, then we can

define convolution operation '$*$' as:

$$x * y = F^{-1}\left(F(x) \cdot F(y)\right) \tag{8}$$

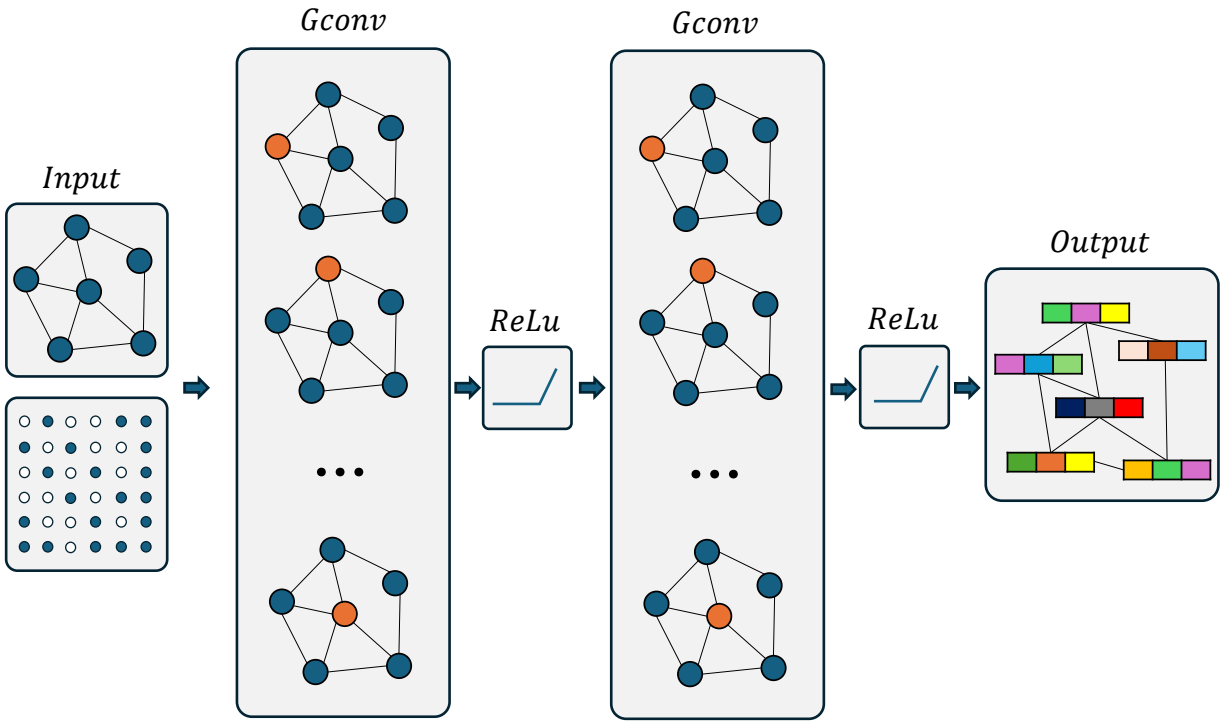

Figure 3: Graph Convolution Network

Graph Convolutional Network (GCN) is the blueprint of what a GNN looks like. Introduced by Kipf and Welling in 2017, it is based on the concept of an efficient variation of Convolutional Neural Networks (CNNs) used in graph counterparts. Since their inception, GCNs have quickly become the most prevalent architecture in the family of GNN architectures. As groundbreaking as it is, 'vanilla' GNN has its downsides. A conventional GNN layer cannot differentiate between nodes with an unbalanced number of neighbors. Consider the following example: node A has 2000 neighbors, but node B only has two. The GNN layer would embed these nodes based on $h_i = \sum_{j \in N_i} x_j \; W^T$, which is unfair to node B as $h_A \;>>\; h_B$ owing to the large amount of neighboring nodes that A possesses. A simple method is to divide h_i by a normalization coefficient $deg(i)$, which represents each node's degree (number of neighbors).

How do we interpret the normalizing coefficient in terms of matrix multiplication? Let D be a diagonal matrix that represents the degree of each node on the diagonal. By the definition of the matrix D, it provides the factor $deg(i)$. Therefore, the normalization coefficient $\frac{1}{deg(i)}$ is given by the inverse of D, i.e., D^{-1}. To add the self loops, we add identity matrix, I, with D to obtain $\tilde{D} = D + I$. Hence, $\tilde{D}^{-1} = (D\ +\ I)^{-1}$ is our normalization coefficient. In vanilla GNN, $H = \tilde{A}\ X\ W^T$, where $\tilde{A} = A\ +\ I$, allowing for self-loops. Incorporating the normalization coefficient, $H = \tilde{D}^{-1}\ \tilde{A}\ X\ W^T$ normalizes the rows. In node-level, the normalization translates to $h_i = \frac{1}{deg_i} \sum_{j \in N_i} x_j\ W^T$.

The authors of the article, Kipf and Welling, discovered that node properties spread quicker in nodes with higher degrees than in nodes with lower degrees. To address this, they developed a hybrid normalization technique:

$$H = \tilde{D}^{\frac{-1}{2}}\ \tilde{A}\ \tilde{D}^{\frac{-1}{2}}\ X\ W^T \tag{9}$$

This is also called the symmetric normalization. In node-level, the hybrid normalization translates to $h_i = \sum_{j \in N_i} \frac{1}{\sqrt{deg_i\ deg_j}} x_j\ W^T$. This leads us to the following layer-wise propagation equation:

$$H^{(l+1)}\ =\ \sigma\ \left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{10}$$

where σ represents activation function such as ReLU, $H^{(l)}$ represents layer-specific activation matrix with $H^{(0)}$ denoting X, $W^{(l)}$ similarly represents layer-specific weight matrix.

¡here comes model architecture diagram¿

2.4 GraphSAGE

The process of learning continuous vector notations, also known as the graph's embeddings, is referred to as representation learning on graphs. The objective is to create low-dimensional vector representations that preserve significant attributes of the nodes and their relationships while encoding the structural and relational information found in graphs. The two different approaches of learning these embeddings are inductive and transductive. Trans-

ductive learning focuses on learning the embeddings optimized specifically for the nodes in the training set. On the other hand, inductive learning tries to generalize a model that predicts the embeddings for unseen datapoints. Various existing representation learning methods are primarily transductive, one excellent inductive learning approach proposed by Hamilton et. al is GraphSAGE.

The fancy acronym for "Graph Sample and Aggregate" is GraphSAGE. As the name implies, the two main functions of this framework are node sampling from the vicinity of the center node and feature aggregation from the sampled nodes. As it is an inductive learning approach, instead of training individual embeddings on each node from the graph, it learns a parameterized function that generates embeddings by the two primary operations — sample and aggregate. The sampling is done from nodes from k-hop neighborhoods of the center node. Each node from the sampled node set are then iterated over their neighbors for the chosen k-hop.

The algorithm for the forward propagation, i.e, embedding generation is as follows:

Consider a graph $G(V, E)$ with node features $\{x_v, \forall v \in V\}$. Let K denote the depth of the graph (neighborhood) and $W^k, \forall k \in \{1, \ldots, K\}$ denote the weight matrices in specific k^{th} hop. Let N denote the neighborhood function of v and σ be the non-linearity introduced during aggregation of the outputs from the aggregation function, $AGGREGATE_k, \forall k \in \{1, \ldots, K\}$.

1. **Initialize:** $h_v^0 \leftarrow x_v, \forall v \in V$
2. Iterate upon k neighborhoods: **for** $k = 1 \ldots K$ **do**:
   (a) Iterate upon v vertices: **for** $v \in V$ **do**:
      i. Aggregate the features from the previous neighborhood nodes:
      $h_v^k \leftarrow AGGREGATE_k(\{h_u^{k-1}, \forall u \in N(v)\})$
      ii. Introduce non-linearity to the weighted linear transformation:
      $h_v^k \leftarrow \sigma(W^k \cdot CONCAT(h_v^{k-1}, h_{N(v)}^k))$

(b) Normalize h_v^k by L2 regularization.

3. **Output:** $z_v \leftarrow h_v^K, \forall v \in V$

The commonly aggregation functions are MEAN and POOL aggregators which are trainable and symmetric i.e., the order of the nodes in the operations do not matter. MEAN aggregators aggregate the node features as per the algorithm discussed above and computes the mean of the sampled nodes' features.

$$h_v^k \leftarrow \sigma\left(W \cdot MEAN\left(\{h_v^{k-1}\} \cup \{h_u^{k-1} \forall u \in N(v)\}\right)\right) \tag{11}$$

POOL aggregator is transformation function which applies element-wise max-pooling to the node features.

$$AGGREGATOR_{pool}^k = max\left(\{\sigma\left(W_{pool} h_u^k + bias\right), \forall u \in N(v)\}\right) \tag{12}$$

GraphSAGE is a useful tool for producing embeddings for unseen nodes. They are versatile and employed at different phases of prediction problems because of their representation learning application.

Some Application of GraphSAGE:

1. A few members of IEEE, J Liu et al. have used GraphSAGE to forecast traffic speeds in addition to a data recovery algorithm for missing data. In the paper titled "GraphSAGE-Based Traffic Speed Forecasting for Segment Network With Sparse Data", they proposed an algorithm which imputes missing data with the correlations drawn from spatial and temporal data. Then they use GrapSAGE to forecast the traffic speeds within the studied road network.

2. In a paper titled "Causal GraphSAGE: A robust graph method for classification based on causal sampling", an advancement in sampling technique is proposed by Zhang et al. in 2022. It introduces causal sampling method instead of the unbiased sampling used in GraphSAGE. In addition to focusing on the structure of the target node it

focuses on the structure of the neighbors too, making the node embeddings robust for classification.

3. Weng Lo et al. propose E-GraphSAGE, a new network intrusion detection system which uses GNNs to leverage the inherent structure of graphs and demonstrate the potential of GNNs for IOT network intrusion detection systems in their paper titled "E-GraphSAGE: A Graph Neural Network based Intrusion Detection System for IoT".

2.5 Graph Attention Networks

Graph Attention Networks (GATs) draw inspiration from the attention mechanism introduced by Bahdanau et al. in 2015, introducing a paradigm shift in graph neural networks. To comprehend the significance of GATs, it is illuminating to consider a real-world example. Imagine a person, A, with 10 friends, 4 of whom are deemed good while the rest are categorized as bad. Traditional graph neural network architectures like vanilla GNNs or GCNs often make classification decisions based solely on the count of good and bad friends. This simplistic approach may lead to misclassifications, especially when the proximity to certain friends is not considered. For instance, if person A is significantly closer to their good friends than the bad ones, a conventional model might erroneously categorize A as bad due to the sheer count. The crux of the matter lies in the need to assign importance to nodes within each neighbourhood, accounting for the relational nuances. Enter graph attention networks to address this challenge.

GATs fundamentally recognize that not all nodes are equal; some hold more significance than others. Unlike GCNs, which employ static normalization coefficients, GATs propose dynamic weighting factors determined through a process known as self-attention. This self-attention mechanism is akin to that employed in transformer architectures, allowing GATs to adaptively assign weights to nodes based on their contextual importance within a given neighbourhood. This nuanced approach enhances the discriminative power of GATs, enabling them to capture intricate relationships and dependencies crucial for accurate graph-based predictions. In essence, GATs introduce a layer of graph processing that pays attention to the inherent importance of each node, mitigating the limitations of traditional

architectures and fostering more nuanced and accurate graph-based learning.

Now, delving into the mathematical underpinnings of graph neural networks, we uncover a comprehensive process comprising four key steps in calculating attention scores: Linear transformation, activation function application, SoftMax normalization, and multi-head attention. Let's dissect each of these steps:

1. **Linear transformation :**

   (a) Initially, let's designate the node under focus as node i and select its neighbor j for computation.

   (b) The first step in linear transformation layer is to transform our input node features(x_i,x_j) into higher level features.

   (c) To achieve this, we utilize a trainable weight matrix W which is then multiplied with the node features of the given node and its neighbors, yielding Wx_i and Wx_j.

   (d) Once this transformation is done the higher level features are concatenated.This is expressed as $[Wx_i \,||\, Wx_j]$.

   (e) The second step is applying an additional linear transformation W_{att} on this concatenated vector to produce the attention coefficients a_{ij}.

   (f) Therefore we can say that $a_{ij} = W_{att}([Wx_i \,||\, Wx_j])$.

   (g) The presence of multiple weight matrices aids in introducing a learnable factor, crucial for inducing adaptability.

   (h) These attention coefficients portray the significance of node j to node i, setting the stage for the subsequent activation function application.

2. **Activation function :**

   (a) Non-linearity is fundamental in neural networks for capturing intricate data details.

   (b) Here, the activation function plays a pivotal role. We apply a non-linear activation function to the previously computed attention coefficients.

(c) In contrast to the standard ReLU function prone to the dying ReLU problem, we opt for the leaky ReLU, offering a remedy to this issue.

(d) Thus, we express this process as $e_{ij} = \text{LeakyReLU}(a_{ij})$

3. **SoftMax Normalization :**

(a) Neural networks frequently meet disparities in the ranges of these attention score values due to data volatility, complicating comparability.We don't want to end up in situations where the magnitude of a few scores overflows our entire function.

(b) To mitigate this issue, we normalize these scores using a softmax function, making sure all the scores aggregated together outputs 1.

(c) Mathematically, this normalization is depicted as

$$\alpha_{ij} = softmax(e_{ij}) = \frac{exp(e_{ij})}{\Sigma_{k \in N_i} exp(e_{ik})} \tag{13}$$

4. **Multi - head attention :**

(a) While the aforementioned steps pertain to self-attention, a stability challenge arises in this mechanism.

(b) To address this, we introduce multi-headed attention, stabilizing the process and enabling parallel implementation.

(c) This involves iteratively repeating the above steps across multiple attention heads.

(d) In hidden layers, the resulting features are concatenated, while in the output layer, an average is computed.

(e) So here are a few notations :

- This is how the aggregated features from the k^{th} head for node i is going to appear

$$h_i^k = \sum_{j \in N_i} \alpha_{ij} W x_j \tag{14}$$

- When employing multiple heads, multiple feature vectors are obtained, concatenated in hidden layers, expressed as

$$h_i = ||_{k=1}^{n} h_i^k = ||_{k=1}^{n} \Sigma_{j \in N_i} \alpha_{ij} W x_j \tag{15}$$

- In the output layer, averaging is performed instead of concatenation, resulting in the equation

$$h_i = \frac{1}{n} \sum_{k=1}^{n} h_i^k = \frac{1}{n} \sum_{k=1}^{n} \sum_{j \in N_i} \alpha_{ij} W x_j \tag{16}$$

Some applications of Graph Attention Networks :

- In 2023, researchers from China published a paper titled "Static Voltage Stability Margin Prediction Considering New Energy Uncertainty Using Graph Attention Networks and Long Short-Term Memory Networks." Their objective was to develop a predictive model leveraging Graph Attention Networks (GATs) and Long Short-Term Memory Networks (LSTMs) to swiftly and accurately forecast static voltage stability margins. This model aids in precisely evaluating voltage stability states and enables timely implementation of control measures to maintain stable operation in power systems.

- Dan Busbridge et al. introduced Relational Graph Attention Networks as an extension of GATs tailored for predictive tasks on heterogeneous datasets like AIFB and MUTAG.

- Additionally, Ziming Wang et al. presented Edge-featured Graph Attention Networks in the International Conference on Artificial Neural Networks. This model incorporates edge features, unlike basic GAT models that primarily consider node features. Through experiments on benchmark datasets such as PubMed, CiteSeer, and Cora, the researchers demonstrated significant improvement over the original GAT model.

2.6 Graph Isomorphism Networks

Before delving into Graph Isomorphism Networks (GINs), it's essential to grasp the concept of expressiveness within the realm of graph algorithms. Expressiveness denotes

the capacity of a specific network to aptly capture intricate relationships and discern various graph structures. In the context of Graph Neural Networks (GNNs), expressiveness is pivotal, as it signifies the ability to learn and represent diverse features and attributes of nodes and edges within a graph effectively. The ultimate objective of any GNN is to generate optimal embeddings, ensuring that disparate nodes possess distinct embeddings while similar nodes share analogous ones. The process of computing node embeddings hinges on leveraging node features and the connections within the graph. These embeddings serve as unique identifiers for each node, encapsulating its characteristics comprehensively. The generation of embeddings forms the bedrock for a myriad of classification tasks achievable on the graph, facilitating predictive analyses and fostering deeper insights into graph structures.

As we delve deeper, it's essential to understand how these node embeddings are computed and how nodes are categorised based on their similarities or differences. The mechanism underlying this process is analogous to that employed in preceding network architectures. Essentially, node embeddings stem from a fusion of node features and their connections, ensuring a unique representation for each node and enabling accurate discernment of differences and similarities among them. The construction of these embeddings typically involves two sequential steps: aggregation and an update function. During aggregation, neighbouring nodes considered by the GNN are selectively identified, while the update function combines embeddings from these nodes with the target node's embedding to yield the final embedding. These functions are instrumental in determining the network's expressiveness, impacting its ability to capture nuanced graph structures effectively.

As we explore the computation of node embeddings, it's evident that the mechanism plays a crucial role in determining the network's expressiveness. However, prior architectures have shown limitations in effectively capturing nuanced graph structures due to simplistic aggregation methods like mean, max, or min aggregators.For instance, the widely used Graph Convolutional Network (GCN) employs mean aggregation, which fails to discern nuanced differences in node neighborhoods. This deficiency becomes apparent when considering nodes with varying compositions of neighbour types. For example, GCN cannot recognise the difference between a node having 10 neighbors of type A and 10 neighbors of type B versus

a node with 2 neighbors of type A and 2 neighbors of type B. This issue arises because all node features are normalized during the aggregation process. Similarly, other aggregation methods like max or min aggregators tend to overlook crucial neighborhood structures, thus hindering their ability to capture essential node information effectively. Addressing this challenge necessitates the adoption of an injective function that maps distinct embeddings to unique outputs, ensuring a comprehensive representation of graph structures.

In response to this imperative, Graph Isomorphism Networks (GIN) emerged as a specialized class of Graph Neural Networks (GNNs), first introduced by Xu et al. in their seminal paper "How Powerful are Graph Neural Networks?" (2018). GIN architecture caters primarily to domains where graph expressiveness holds paramount importance, notably in tasks like graph classification. Drawing inspiration from the Weisfeiler and Lehmann test for graph isomorphism, GIN adopts a sum aggregation strategy to retain neighborhood information. Subsequently, this aggregated information undergoes processing by a Multilayer Perceptron (MLP), leveraging the Universal Approximation Theorem to ensure the function's injectiveness. The Universal Approximation Theorem asserts that "A neural network with at least one hidden layer and a non-linear activation function can approximate any continuous function to arbitrary accuracy."

Next let's delve into the concept of the Weisfeiler-Lehman (WL) test, a fundamental algorithm proposed by Weisfeiler and Lehman in 1968 to determine graph isomorphism efficiently. The WL test aims to establish the canonical form of a graph, allowing for straightforward comparison between graphs to ascertain their isomorphism.The WL test proceeds as follows:

1. Initially, each node in the graph is assigned the same color.
2. Each node aggregates its own color with the colors of its neighboring nodes.
3. The resulting aggregated colors are then processed through a hash function, generating a new color.
4. Subsequently, each node aggregates its new color along with the new colors of its neighboring nodes.

5. The resulting aggregated colors undergo further processing through a hash function to produce updated colors.

6. These steps iteratively continue until no further changes in node colors occur.

This iterative process ensures that the colors assigned to nodes capture the structural information of the graph effectively. By comparing the canonical forms derived from this process, the WL test enables efficient determination of graph isomorphism.This rigorous methodology serves as a foundational component in various graph-related tasks, providing a robust mechanism for assessing the structural similarities between graphs. Next, we'll look at the math behind computing node embeddings in general, as well as how GIN does it.

- General equation for calculating node embeddings can be expressed as follows:

$$h_i^{'} = \phi(h_i, f([h_j : j \in N_i])) \tag{17}$$

  Explanation : The function f determines the neighboring nodes that the Graph Neural Network (GNN) considers, while the combining function ϕ integrates the embeddings from these selected nodes with the target node's embedding, resulting in the creation of the updated embedding for the target node.

- For GINs the equation looks something like this:

$$h_i^{'} = MLP((1+\epsilon).h_i + \sum_{j \in N_i} h_j) \tag{18}$$

  Explanation : In Graph Isomorphic Networks (GINs), the aggregation process begins by combining all neighborhood embeddings with the target node's embeddings. The authors introduce an ϵ parameter, serving as either a learnable parameter or a fixed scalar that signifies the significance of the target node. Subsequently, an MLP (Multi-Layer Perceptron) is employed as the injective function to derive the final embeddings.The benefit with using neural networks is that over the course of training the appropriate injective function can be learnt.

Some applications of Graph Isomorphism Networks :

- Graph Isomorphism Networks (GINs) are widely renowned for their efficacy in graph classification tasks, particularly on benchmark datasets like those containing protein structures or IMDb-B data. GINs have emerged as a go-to choice due to their ability to effectively analyze and classify graphs, making them invaluable tools in various research domains.

- In 2020, Yuzhong Peng et al. introduced an enhanced variant of Graph Isomorphism Network (GIN) named MolGIN, specifically tailored for predicting molecular ADMET (Absorption, Distribution, Metabolism, Excretion, and Toxicity) properties. This innovative approach addresses the crucial task of assessing the ADMET properties of molecules, which holds significant importance across various domains such as drug development, industrial chemicals, agrochemicals, cosmetics, environmental science, and food chemistry. By leveraging MolGIN, researchers can effectively predict and evaluate key molecular properties, contributing to advancements in pharmaceutical research and related fields.

2.7 Variational Graph Auto - Encoders

To comprehend the concept of Variational Graph Auto-Encoders (VGAE), it's essential to grasp the functionalities and applications of auto-encoders, variational auto-encoders (VAEs), and graph auto-encoders (GAEs).

Auto - encoders as the name suggests has an encoder and a decoder. The encoder is tasked with transforming an input X into a lower-dimensional embedding Z, also termed as a latent space representation. Subsequently, the decoder reconstructs this embedding to produce an output $\hat{X}$. The formulation of the loss function is tailored to the specific objectives of the auto-encoder. These models find utility across various domains such as image compression, denoising, anomaly detection, feature extraction, and dimensionality reduction.

An inherent limitation of traditional auto-encoders lies in their inability to facilitate gen-

erative tasks, primarily due to the lack of a mechanism for sampling from the latent space embeddings. In response to this challenge, variational auto-encoders (VAEs) were developed to enable the generation of new data from an existing dataset. Unlike conventional auto-encoders that map the input X to a single point in the latent space, VAEs instead embed it into a distribution, typically a Gaussian distribution, from which samples can be drawn. This distribution is parameterized by the encoder to predict both the mean and standard deviation. Notably, the loss function of VAEs comprises two key components: the reconstruction loss, which ensures the fidelity of the reconstructed output $\hat{X}$ to the original input X, and the Kullback-Leibler (KL) divergence loss. The latter acts as a regularizer, constraining the learned distribution to remain close to a prior distribution. Various applications of VAEs include image generation, interpolation, and recommendation systems.

Both graph auto-encoders and graph variational auto-encoders were introduced by Kipf and Welling in 2016. The graph auto-encoder, akin to the original auto-encoder, comprises two components: the encoder and the decoder. The encoder typically employs graph convolutional networks (GCN), graph attention networks (GAT), or other graph-based layers, which take input features X and the adjacency matrix A to generate the embedding Z. The decoder, in contrast, is simpler and involves taking the embedding Z as input, computing a dot product with its transpose, and applying a sigmoid function to introduce non-linearity. This process can be succinctly expressed as:

$$Z = GCN(X, A) \tag{19}$$

$$\hat{A} = \sigma(Z.Z^T) \tag{20}$$

Following the generation of the reconstructed adjacency matrix, it is passed into a loss function to compute the discrepancy between the calculated adjacency matrix and the original adjacency matrix. Typically, this involves employing a binary cross-entropy (BCE) loss function.

Having laid the groundwork with an understanding of autoencoders, variational autoencoders (VAEs), and graph autoencoders (GAEs), we can now delve into graph variational

autoencoders (VGAEs). Much like the distinction between autoencoders and VAEs, the difference between GAEs and VGAEs lies in their approach to learning node embeddings. While GAEs directly learn node embeddings, VGAEs take a probabilistic approach by learning normal distributions from which embeddings are sampled. The architecture comprises an encoder module and a decoder module, akin to its predecessors. The encoder utilizes a graph convolutional network (GCN) or a graph attention network (GAT) to learn the mean and standard deviation of the normal distribution in the latent space, which represents the compressed and abstract version of the input data. Subsequently, the decoder samples embeddings from the latent space distribution and calculates the inner product between the latent variables to approximate the adjacency matrix, mirroring the functionality seen in graph autoencoders. Analogous to the loss function in VAEs, VGAEs incorporate a reconstruction loss and a Kullback-Leibler (KL) divergence loss. The primary applications of graph variational autoencoders encompass graph generation and link prediction, among others.

2.8 Relational Graph Convolution Networks

Heterogeneous graphs, characterized by nodes and edges of varying types, present unique challenges due to their diverse and interconnected nature. Similarly, knowledge graphs (KGs) encode rich semantic relationships between entities, providing a structured representation of factual knowledge. KGs are used widely for its structured way of representing information using triples of subject-predicate-object. For instance, Biden - President of - United States is an example of a triple in a KG.

Among the diverse array of GNN architectures, Relation Graph Convolutional Networks (RGCNs) stand out as a particularly promising approach for handling heterogeneous or knowledge graphs. A guiding framework for developing expressive representations that effectively capture the intricate relationship structure seen in heterogeneous or knowledge graphs is provided by RGCNs. RGCNs make reasoning, inference, and prediction operations on these graph structures more efficient by explicitly describing various relations and interactions between elements.

Before getting into the working and the intricacies in the RGCN model, let's represent the graph using the triplets of subject-predicate-object. Let the graph be denoted as:

$$G = (V,\ E,\ R) \tag{21}$$

where:

1. V represents the set of nodes or entities, i.e., any node $v_i \in V$
2. E denotes the set of triples represented as $(v_i,\ r,\ v_j)$
3. R denotes the set of relation types. In (2), $r \in R$

RGCN, as its name implies, is a graph convolution network extension for relational graphs. Drawing inspiration from GCNs' efficient encoding of local, structured neighborhood information in graphs, the RGCN model considers an extra relational aggregate for nodes in relational multi-graphs. The layer-wise learning is modeled by the equation:

$$h_i^{(l+1)} \ = \ \sigma\left(W^{(l)}h_i^{(l)} \ + \sum_{r\in R}\sum_{j\in N_i^r} \frac{1}{deg(i)_r}\ U_r^{(l)}h_j^{(l)}\right) \tag{22}$$

The T_2 term of the equation is slightly different from the equation of GCN as RGCNs take the type of relations $r \in R$ as an aggregator of neighborhood nodes.

- $h_i^{(l)}$ denotes the representation of node i at layer l,
- N_i^r represents the set of neighboring nodes of node i under relation type r,
- $W^{(l)}$ and $U_r^{(l)}$ denote the learnable weight matrices for the self-connection and relation-specific connection respectively,
- $deg(i)_r$ is a normalization factor representing the degree of nodes connected by relation r to node i,
- σ represents the activation function.

One concern to be noticed from the equation is the involvement of $U_r^{(l)}$, the learnable weight matrix for relation-specific connection. As the number of relations in a multi-relational graph increases rapidly, the number of learnable parameters increase rapidly too. This leads to overfitting the model on those relations that are lesser in number. Hence, there is a need for regularization of $U_r^{(l)}$. The authors, Schlichtkrull et. al suggest two approach to regularization.

1. **Basis Decomposition**

   Basis decomposition helps in associating each relation type with an importance factor. The learnable weight matrix is written as a linear combination of basis matrices V_b with importance factors denoted by α_{rb}.

$$U_r^{(l)} = \sum_{b-1}^{B} \alpha_{rb}^{(l)} \, V_b^{(l)} \tag{23}$$

   where B denotes the no. of bases and $V_b^{(l)} \in R^{d(l+1) \times d(l)}$.

2. **Block Diagonal Decomposition**

   A block diagonal matrix is a square matrix of the form:

$$\begin{bmatrix} A_1 & & 0 \\ & \ddots & \\ 0 & & A_m \end{bmatrix} \tag{24}$$

   where $A_1, \cdots, A_m$ are square matrices with values lying on the diagonal and all the entries of the matrix equal 0.

   Block decomposition of $U_r^{(l)}$ means that we represent it as direct-sum of matrices, $Q_{1r}^{(l)}, \cdots, Q_{Br}^{(l)}$ which makes these matrices to be in the diagonal of the matrix $U_r^{(l)}$.

This brings us to represent the weight matrix as:

$$U_r^{(l)} = \bigoplus_{b=1}^{B} Q_{br}^{(l)} \tag{25}$$

Block decomposition helps in bringing in the sparsity constraint to the learnable parameters.

While both the regularization methods help in reducing the no. of learnable parameters, basis decomposition is proposed to specifically eliminate the overfitting on rarer relations by sharing the parameters between relations which are both larger and smaller in number. RGCNs are specifically equipped for node classification and link predictions on KGs.

Some Application of RGCNs:

2.9 Evolve GCN

Temporal graphs, also referred to as time-varying graphs, encompass the dynamic nature of relationships between entities, offering a robust framework for modeling and analyzing time-varying dependencies. Unlike static graphs that capture a snapshot of relationships at a single time stamp, temporal graphs portray evolving structures where nodes and edges change over time. Broadly categorized into static temporal and dynamic temporal graphs, these representations vary in the extent of change observed across time.

Static temporal graphs maintain a consistent topology over time, with node features evolving while fundamental graph structures such as the number of nodes and edges remain unchanged. This scenario is prevalent in applications like traffic forecasting, where the underlying network structure remains constant while attributes of nodes (e.g., traffic flow) vary temporally.

In contrast, dynamic temporal graphs witness fluctuations in both node features and graph topology over time. Here, the entire structure of the graph undergoes modifications across different time snapshots, with variations in the number of nodes, edges, and node

attributes. This dynamic nature is exemplified in social networks, where connections between users can emerge or dissolve over time, requiring models capable of capturing these evolving interactions.

The task at hand involves making predictions on node features over time using graph neural network (GNN) models. While traditional GNNs such as Graph Attention Networks (GAT), Graph Convolutional Networks (GCN), and Graph Isomorphism Networks (GIN) excel at node-level predictions on static graphs, they lack the ability to incorporate temporal dependencies across multiple time snapshots. To address this, we turn to recurrent models inspired by traditional neural networks like Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Gated Recurrent Units (GRUs). Specifically, we delve into Evolutionary Graph Convolutional Networks (Evolve GCN), a novel architecture that integrates LSTM or GRU cells to leverage temporal information and learn evolving node representations over time. This approach enables the model to retain memory of previous snapshots while computing node features for subsequent time steps, facilitating more comprehensive temporal analysis and prediction tasks in dynamic graph settings.

Evolve GCN, initially proposed by Pareja et al. in 2019, is a novel architecture that combines a Graph Convolutional Network (GCN) with a recurrent neural network (RNN) layer, such as a Gated Recurrent Unit (GRU) or Long Short-Term Memory (LSTM) network. The GCN layer, akin to a perceptron but with an additional neighborhood aggregation step, generates node embeddings based on the given adjacency matrix and node embedding matrix. So the math behind GCN can be quickly summarised below:

At a given time t the l^{th} layer of the GCN takes the adjacency matrix A_t and the node embedding matrix H_t^l , and uses a weight matrix W_t^l to update the node embedding matrix to H_t^{l+1} as output. Mathematically we can write this as:

$$H_t^{l+1} = GCN(A_t, H_t^l, W_t^l) \tag{26}$$

Once this is done, it is imperative to perform weight evolution which is were the use of recurrent neural networks come in. The update of the weight matrix W_t^l at time t and

layer l is based on the current as well as historical information. When employing a Gated Recurrent Unit (GRU), denoted as Evolve GCN-H, the current GCN parameters and current node embeddings are passed to the GRU to generate the updated GCN parameters:

$$W_t^l = GRU(H_t^l, W_t^{l-1}) \tag{27}$$

On the other hand, choosing Evolve GCN-O involves utilizing a LSTM where only the previous GCN parameters are passed inside the LSTM layer to generate the current GCN parameters. There is no need to pass the current node embeddings as in the GRU-based variant. This formulation is represented as:

$$W_t^l = LSTM(W_t^{l-1}) \tag{28}$$

The selection between Evolve GCN-H and Evolve GCN-O depends on the characteristics of the dataset. When the node features contain valuable information, opting for the Evolve GCN-H version might be more beneficial. Conversely, if the node features are less informative, the Evolve GCN-O version may yield more effective results. This consideration underscores the importance of tailoring the model architecture to the specific dataset characteristics to achieve optimal performance.

2.10 Attention Temporal Graph Convolution Network [A3TGCN]

In the evolving landscape of temporal graph analysis, where nodes and edges dynamically shape connections over time, models like Evolve GCN has emerged as a potent tool for uncovering underlying patterns and insights. Leveraging temporal information encoded within graph structures, Evolve GCN showed remarkable efficacy in predictive tasks like link prediction over a period of time. However, as the complexity and scale of temporal graph data continues to increase there arises a need for enhanced models capable of capturing intricate temporal dependencies while mitigating computational overhead.

In response to this challenge, Attention Temporal Graph Convolution Network (A3TGCN) was introduced. It brings in soft attention mechanism which reduces the computation by accounting to important timestamp data using attention score and context vector. Having

previously explored the motivation behind combining traditional GCNs with recurrent units like GRUs or LSTMs, we will look into the feature that A3TGCN brings in, which is the attention mechanism. Simply said, A3TGCN is a combination of GCN, GRU and attention.

In A3TGCN, GCNs are employed to extract features from the graph structures, enabling the model to capture spatial relationships between nodes and edges. GRU modules facilitate the modeling of temporal dependencies, allowing the network to adaptively learn and update its representations as the graph evolves. Attention mechanisms are integrated to dynamically weigh the importance of different nodes and edges at each time step, enhancing the model's ability to focus on relevant information and improve predictive performance.

Firstly, let's uncover the GCN module. GCN as seen in equation (9) can be seen as a function on parameters A and X denoting the adjacency matrix and node features respectively. Here, we use a two layer GCN and the GCN operation can be denoted as:

$$GCN(A, X_t) = \sigma\left(\tilde{A}\ \sigma\left(\tilde{A}\ X_t\ W_0\right)\ W_1\right) \tag{29}$$

where X_t denotes the feature set at time t.

Secondly, GRU module comes into picture. GRU contains two gates namely update gate and reset gate and their outputs are denoted here by u_t and r_t, respectively at time t. The outputs of the gates are used in finding the candidate hidden state $h_t^{'}$. Let h_t represent the final hidden state value at time t. The equations are as follows:

$$u_t = \sigma\left(\ W_u\ [GCN(A, X_t),\ h_{t-1}] + b_u\ \right) \tag{30}$$

$$r_t = \sigma\left(\ W_r\ [GCN(A, X_t),\ h_{t-1}] + b_r\ \right) \tag{31}$$

where W and b denote the weight matrices and biases in each state. The calculated $h_t^{'}$ and u_t are used to calculate the hidden state, h_t as:

$$h_t = (1 - u_t) * h_t^{'} + u_t * h_{t-1} \tag{32}$$

Lastly, attention mechanism is used. The hidden states from GRU, $\{h_{t-n}, \cdots, h_{t-1}, h_t\}$, are passed into a Multi-Layer Perceptron (MLP) using Softmax to find the weights corresponding to each hidden states called the attention scores. The attention scores, $\{\alpha_{t-n}, \cdots, \alpha_{t-1}, \alpha_t\}$, are the factor of importance associated to each hidden state. The context vector C_t is computed as the weighted sum of the hidden states with their attention scores as:

$$C_t = \sum_{i=1}^{n} \alpha_i * h_i \tag{33}$$

Further, a fully-connected layer is used for predictions with respective loss function for which C_t acts as a parameter.