Graphical Neural Networks: Learning on Graph-Structured Data

Master of Engineering
Information Technology
Akanksha Venkatesh Venkatesh Baitipuli
1568316
akanksha.venkatesh-baitipuli@stud.fra-uas.de

***Abstract*— Graph Neural Networks (GNNs) are frameworks that develop deep learning algorithms for graph-structured data by enabling each node in the graph to learn features about adjacent nodes through message passing and feature aggregation. This paper presents the theory of Graph Neural Networks (GNNs), including graph convolution operations, the message-passing framework, and critical properties such as permutation invariance, along with spectral and spatial interpretations of graph convolution. The Key architectures of GNNs, such as Graph Convolution Networks (GCN) and Graph Attention Networks (GATs), as well as invariance properties, applications, and challenges faced by GNNs, concerning the Weisfeiler-Lehman test, are discussed. This paper provides an overview of the theoretical and practical aspects of GNNs, focusing on message passing mechanisms and graph convolution operations. , and then demonstrate it with a proof-of-concept implementation of a two-layer Graph Convolutional Network (GCN) for a sensor network of automotive Electronic Control Units (ECUs).**

Keywords*— *Graphical Neural Network(GNNs), Deep learning, Graph Convolution Networks (GCNs), Graph Attention Networks(GATs), Sensor Networks, Electronic Control Units(ECUs), Fault detection, Automotive Systems, Node classification.

I. INTRODUCTION

In today's world of intelligent systems, the advancements of artificial intelligence (AI) into real-world applications have increased rapidly, along with the handling of structured and unstructured data with the development of deep learning. Other neural networks, like Convolutional Neural Networks (CNNs), perform well in image-related tasks and Recursive Neural Networks (RNNs) with sequential data processing, but traditional deep learning models face difficulties with non-Euclidean data structures such as graphs.

Many real-world scenarios, such as molecular graphs, social networks, and sensor systems in automobiles, can be represented as graphs. Unlike structured data like sequences or images, graphs demonstrate non-Euclidean structures with arbitrary sizes, uneven neighborhood connectivity, and a lack of a fixed ordering among nodes. This led to the emergence of Graph Neural Networks (GNNs), which are specially designed for processing graph-structured data by enabling nodes to learn the feature representations via message passing and neighborhood aggregation.

In this paper, I will focus on graph neural networks, initially on their theoretical foundations. I begin by discussing GNNs in depth as a solution to modeling structured relationships through nodes and edges in a graph. I also discuss core concepts like message passing frameworks, spectral and spatial graph convolutions, permutation invariance, and the Weisfeiler-Lehman test. I have also presented a proof of concept concerning node classification and explained in detail the code flow with analysis and results.

II. GRAPH NEURAL NETWORK

A Graph Neural Network (GNN) is a neural network model that works with structured data. The nodes are denoted as entities, and the edges as relationships between those entities [1].GNNs use node-level, edge-level, and graph-level information to perform prediction tasks [2]. These networks can be used to simulate large systems such as social networks. GNNs represent graphs, with edges representing dependency information.

GNNs use two different functions: a transition function and an output function. The transition function expresses the relationship between nodes, while the output function particularly tells about the output for each node [3]. Each concept is defined by its features and related concepts. GNNs use a local computational framework with processing done on each node, with communication limited to neighboring nodes [8]. A node's output depends on information from its neighbors [3].

The ability of GNNs to identify influential nodes is valuable in various fields, including biology, computer science, and corporate environments. Social network influencers can be identified using GNNs and GCNs [2]. Overall, GNNs provide a powerful framework for analyzing and extracting insights from graph-structured data in various domains [3].

A. What is a Graph: Nodes, Edges, Adjacency Matrix

A graph *G* is a mathematical data structure that consists of both nodes and edges. Nodes are also called vertices and are denoted as *V,* and edges connecting pairs of nodes are denoted as *E.* Formally, it is defined as $G = (V, E)$ [1]. V-set of nodes (vertices), and E is the set of edges. Figure 1, nodes (depicted as colored circles) [1] represent individual sensors according to my implementation. Each node is assigned a unique node, and different colors are used to indicate different Electronic Control Units (ECUs) to which the sensors belong.

In various real-world applications, nodes can represent users in a social network, devices in a communication system, or sensors in a sensor network. Each node can carry a set of values depending on the problem domain. Where $V= \{v_1,v_2,v_3 \ldots v_n\}$ where $n=|V|$ is the number of nodes in the graph [1]. Each v_i represents an individual entity in the graph for example, a user in a social network or an atom is a molecule.

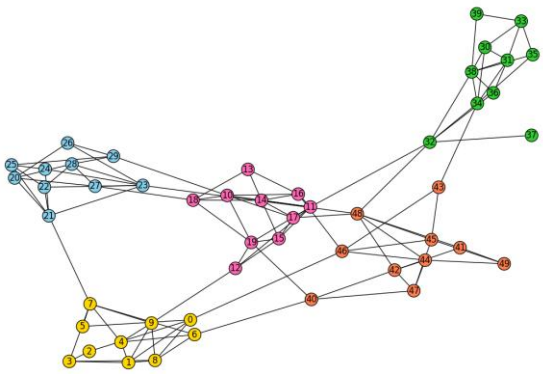


Fig 1: Nodes and edges

Edges define the connections or relationships between the nodes [1]. Edges E = {e_1, e_2,..., e_m} where m=|E| is the number of edges. Each node connects two nodes as (u,v) where u,v∈V in an undirected graph. (u→v) when each edge has a direction. From Figure 1, edges (the lines connecting the nodes) represent communication links or relationships between sensors [1]. Edges (the lines connecting the nodes) show communication links or sensor relationships as shown in Figure 1. An edge between any two nodes represents a direct connection, such as proximity of physical placement, communication capability, or functional interaction. These edges in this show that the visualization are undirected, so the relationships are bidirectional (i.e., if sensor A is connected to sensor B, then sensor B is connected to sensor A).

This graph structure — composed of nodes and edges — is mathematically represented by an adjacency matrix. This matrix records all node-to-node connections. The adjacency matrix, along with the node features, functions as the input to the Graph Neural Network (GNN) model to perform tasks such as fault detection or health monitoring at the sensor or ECU level.

The adjacency matrix is a square matrix that allows us to represent the edges that connect two nodes. The calculated representation of the structure A, where A is the adjacency matrix which belongs to R^{nxn} for a graph where there are n nodes. For instance, consider the nodes as ***i*** and j as an edge connected together, and the entry at position(***i, j***) indicates that the nodes are connected.

If A = 1, then it indicates the presence of a connection, while a value of 0 indicates no connection. As the sensor network here involves mutual communication links, the resulting adjacency matrix is symmetric, representing an undirected graph [2].

$$A_{ij} = \begin{cases} 1 & \text{if there is an edge from node } i \text{ to node } j \\ 0 & \text{otherwise} \end{cases}$$

In this paper, the nodes refer to single sensors, and the edges refer to direct communication lines between them. This relationship is represented using an adjacency matrix, which is one of the essential inputs to the Graph Neural Network (GNN). During training, the GNN utilizes this matrix to regulate the message passing process, where each node gathers information from its neighbors and updates its own representation. This enables the model to learn effectively not just from the sensor's properties but also from connectivity patterns locally stored in the graph.

B. Node Features: How input features are created

Node features can be formulated in various ways depending on the type of data and the specific problem that is being addressed. The two approaches for creating node features are manual annotation and simulation. For example, degree centrality illustrates the level of connectivity among the nodes are represented using bootstrapping techniques. The adjacency matrix, also known as the connection matrix, is commonly used to depict connections between nodes.

For instance, in social networks, node features may include user profile information (e.g., age, location, interests), activity metrics (e.g., number of posts, followers, interactions), or text from posts and comments; in the context of influencer detection, node features may be the number of total interactions each person with a predetermined threshold that supports classification as influencer.

The feature extraction techniques, such as centrality measures (e.g., Closeness, Central importance, Branching, Eigen Centrality) can be used to measure the importance and influence of nodes in the network. The representation of Deep learning techniques creates node embedding's that capture patterns and relationships in the graph structure. The selection and design of these node features determine how well GNNs and other graph-based models work.

C. Graph-Level vs Node-Level Learning Tasks

Node-level learning is mainly a concentration on individual nodes of the graph. It aims to use the unique features of a node and the graph structure for the prediction of the node's attributes, labels, or classification. The node classification task is where a category is assigned to each node, e.g., classifying users in a social network as influencers or non-influencers, and node regression, in which a continuous value, e.g., a user's popularity score, is predicted [7]. Node embedding is another relevant task that deals with learning vector representations that encapsulate the local structural and feature information of the nodes. Node-level learning methods typically involve the use of data from a node and its immediate neighbors in a message-passing procedure to calculate node-specific representations or labels.

Graph-level learning is done on the graph as a whole. The goal is to predict properties or labels for entire graphs rather than nodes. Common applications are graph classification, in which a single label is assigned to the entire graph, e.g., whether a molecule is toxic or not according to its graph representation, and graph regression, where a global feature, such as the overall activity or interaction of a social network, is predicted. Graph embedding is also applied to learn compact representations that summarize the entire graph for further processing. Such operations function by aggregating data from all edges and nodes, typically through pooling or readout layers, to form one global-encompassing representation that maintains the global properties of the graph.

III. The Message Passing Framework

A. Message Passing in GNN

Message passing is a mechanism in GNNs.This process allows the network to learn very difficult patterns from nodes and dependencies within the graph structure.

In GNNs, three steps take place: firstly, compute messages, where each node calculates a message function of the node's current representation. The messages received by a node from its neighbors are aggregated. Common methods of aggregation include summation, where the sum of all incoming messages is the next mean, and the average is taken of all the incoming messages, and then the weights are assigned based on the edge or any other factors. The next step is the node attribution, where this aggregated information is then used to update the node's representation. This update often involves combining the messages with the node's original representation.

The message-passing process can be mathematically represented by an equation[2]:

$$Hv = f(x_v, x_c(v), h_e[v], x_e[v]) \quad (1)$$

where '$x_c(v)$' defines the characteristics of links connecting with 'v' and 'he[v]' represents the embedding of 'v' s' adjacent nodes. The transition function 'f' transforms these inputs into a dimensional space. The connections of the neighbors are aggregated and then used to update the node embeddings.

Node classification tasks are among the many applications where GNNs and GCNs have proven useful. For example, by examining node attributes and connections, they can distinguish influencers from non-influencers in social networks. While GCNs categorize nodes according to their features, a basic GNN model offers a means of expressing prediction tasks at the node, edge, and graph levels 7. Because GNNs can model non-Euclidean spaces, they can be used to analyze graphs with node connections that are defined. Thus, this paper is mainly concerned with node classification. Most node classification methods

B. Message passing mechanism in GNN

The message passing mechanism defines how the information propagates within the graph in GNNs. This happens in three steps that is by message construction, message aggregations, and node state update.

1) Message Construction

Firstly before any information can be propagated, each node must prepare a "message" which shall be sent to the neighboring nodes. The message from node u to v at layer k is given by [4]:

$$m_{\{u \to v\}}^{\{(k)\}} = MSG^{\{(k)\}}\left(h_u^{\{(k)\}}, h_v^{\{(k)\}}, e_{\{u \to v\}}\right) \quad (2)$$

Here $\boldsymbol{h_u^{(k)}}$, $\boldsymbol{h_v^{(k)}}$ are the hidden states of nodes at *k* layer. $\boldsymbol{MSG^{(k)}}$ is a learnable function.

2) Message Aggregation

After the messages are updated, each node aggregates its information from its neighborhood ***N(v).*** The aggregation function shall always be a permutation–invariant function where the order of neighbors doesn't matter since graphs have no inherent node ordering, unlike RNNs the aggregation must be invariant to the neighbors' permutations. So it is represented as [4]

$$h_v^{\{(k)\}} = COMBINE^{\{(k)\}}\left(h_v^{\{(k-1)\}}, a_v^{\{(k)\}}\right)$$

The sum/Mean/Max pooling is given by

$$a_v^{(k)} = \sum_{u \in \mathcal{N}(v)} m_{u \to v}^{(k)} \quad \text{(Sum)}$$

The attention-based aggregation where the weights of neighbors contribute differently

3) Node state update

The last step, where each node combines its previous state $\boldsymbol{h_v^{(k)}}$ with the aggregated messages $\boldsymbol{a_v^{(k)}}$ to produce a new state given by[4] :

$$h_v^{\{(k+1)\}} = UPDATE^{\{(k)\}}\left(h_v^{\{(k)\}}, a_v^{\{(k)\}}\right)$$

Here, the UPDATE $^{(k)}$ is a neural network which is a simple non-linear transformation. Where σ is an activation such as ReLU and $\boldsymbol{W}$ is a learnable weight matrix. For the dynamic state updates, a Gated Recurrent Unit can refine the node states as [4]:

$$a_v^{\{(k)\}} = MAX\ \{\{ReLU\}\left(W \cdot h_u^{\{(k-1)\}}\right), \forall u\ \{N\}(v)\}$$

Graph-structured data is a framework designed for dealing with complex relational systems. This representation is universal, capable of capturing real-world systems, from social networks, where nodes represent users and edges denote friendships or interactions, to biochemical molecules, where nodes correspond to atoms and edges to chemical bonds.

Despite their expressive power, graphs present unique challenges for machine learning. Unlike structured data like images or text, graphs exhibit irregular, non-Euclidean geometries with variable-sized inputs and no inherent node ordering [2]. Additionally, meaningful predictions must be invariant to node permutations—a relabeling of nodes should not alter the output. Capturing long-range dependencies, where distant nodes influence each other through multi-hop connections, further complicates the learning process. Traditional neural networks, such as convolutional neural networks (CNNs) [7] or recurrent neural networks (RNNs), are ill-equipped for such data, as they assume grid-like or sequential structures and fail to preserve relational inductive biases.

At their core, GNNs leverage a message-passing framework, where nodes iteratively aggregate and update their representations based on local neighborhood information[9] .This process inherently respects permutation invariance, as the aggregation order of neighbors does not affect the result. GNNs also employ specialized techniques like graph convolution, which adapts convolutional operations to irregular graphs. Spectral methods, for instance, leverage the graph Laplacian and Fourier transforms to define convolution in the spectral domain, while spatial methods directly aggregate features from neighboring nodes, mimicking localized filters in CNNs[2].

By unifying topological information (encoded in *A*) with node attributes (in X), GNNs learn hierarchical representations that capture both structural and semantic patterns[7]. This ability to operate directly on graph data makes GNNs indispensable for tasks like node classification, link prediction, and graph classification, where preserving

relational context is paramount. The resulting models not only outperform traditional approaches but also provide interpretable insights into complex relational systems, underscoring their transformative potential in fields ranging from social network analysis to drug discovery[1].

IV. The Theoretical Properties and Limitations

A. Permutation Invariance and Equivariance

The fundamental theoretical characteristics of GNNs that operate on graph-structured data are permutation invariance and permutation equivariance. When a network exhibits permutation invariance, its output is independent of the input graph's node and edge order. Formally speaking, the graph-level output (like the prediction for the entire graph) stays the same if you relabeled (i.e., permute) the graph's nodes[4].

When the internal representations of the nodes in the model change in sequence with the input permutation, this is known as permutation variance.These characteristics are important because: Unlike pixels in images, which have fixed locations, nodes and edges in graphs typically have arbitrary identities. Artificial node orderings should not affect any meaningful network model. This guarantees that the structure and properties of the graph, rather than their arrangement in code or memory, will be the sole basis for the model's predictions [4]..

For node-level prediction (e.g., node classification or regression), GNNs must be permutation equivariant, meaning that if the nodes of an input graph are rearranged, the corresponding output features will also be rearranged. Neighborhood aggregation and update in GNNs naturally induce this because at each step, they pool the feature of a node's neighbors and update the node accordingly based on it, following the graph structure and not some particular ordering of nodes [4].

B. Weisfeiler-Lehman Test Relation

A traditional and effective method for determining whether two graphs are structurally identical is the Weisfeiler-Lehman (WL) test. The 1-WL (also known as "color refinement") version uses the multi-set of neighbor labels to iteratively update node labels.[9] To update its embedding, each node iteratively aggregates the representations of its neighbours; this process is closely modelled by the traditional neighbourhood aggregation in GNNs [4].

As a result, the degree to which a GNN can convey an injective mapping from multiple neighbour feature sets to the updated representation, or how "expressive" its aggregation function is, determines how well it can discriminate between various graph structures. If the aggregation function is injective and the scheme runs a sufficient number of iterations, the GNN can theoretically distinguish any pair of graphs that the WL test can [4].

Theoretical research indicates that aggregation-based GNNs are only as good at detecting non-isomorphic graphs as the WL test. In other words, if the WL test cannot distinguish between two graphs, then no standard GNN (such as Graph Convolutional Networks, GraphSAGE, etc.) can [8].

However, GNNs can only match the performance of the WL test if their aggregation and readout functions are injective (one-to-one). Many popular variations of GNNs are less effective at discriminating between graphs because they rely on non-injective functions (like mean or max).

The WL test determines that two graphs are not isomorphic if, after a few rounds, they have different sets of node labels [4]. Two important theoretical characteristics of GNNs that operate on graph-structured data are permutation equivariance and permutation invariance. The node and edge order of the input graph has no bearing on the output of a network that exhibits permutation invariance. Formally, the graph-level output (like the prediction for the entire graph) stays the same if you relabel (i.e., permute) the nodes in the graph.

V. Graph Convolution: Spectral and Spatial

Spectral Convolution

A. Spectral Convolution

Graph Convolutional Networks (GCNs) use spectral theory to apply traditional convolution operations to graph-structured data[7]. The graph Laplacian, which encodes the graph's structure, defines convolution in the spectral domain. As explained above the undirected graph G with n nodes is defined with an adjacency matrix A. The degree matrix D is a diagonal matrix in which each diagonal element denotes the degree of node i.

$$D_{ii}=\sum_j A_{ij} \quad (3)$$

The un-normalized Laplacian matri $L=D-A$ alternatively, the normalized Laplacian matrix is given by

$$L_{norm} = I - D^{-1/2} A D^{-1/2} \quad [5]$$

This normalization ensures that across all nodes with varying degrees the Laplacian behaves consistently, which is an important aspect for stable learning.

In Eigen decomposition and Graph Transform the Laplacian matrix L is positive semi-definite and symmetric, which can be diagonalised using the Eigen vectors [5]:

Given a graph signal $x \in R^n$ for instance, consider a scenario of sensor readings in my project, the graph Fourier transform is given by:

$\hat{x} = U^T x$ And its inverse is given by $x = U\hat{x}$.

A spectral filter $g\theta(\Lambda)$, [8], which is parameterized by θ, operates in the Fourier domain:

$$L=U\Lambda U^T$$

$$g_{\theta *}x=Ug_\theta(\Lambda)U^Tx$$

$g_\theta(\Lambda)$ is a diagonal matrix where $\Lambda ii=\theta i$ where θi is a learnable parameter[8].

The simplified Chebyshev approximation is to avoid the Eigen decomposition. So the Chebyshev polynomials approximation of gθ(Λ) is given by[8]:

$$g_\theta * x \approx \sum_{k=0}^{K} \theta_k T_k(\tilde{L}) x$$

Hence, the practical implications of Localized filters with respect to Spectral CNN are that it learns the filters as $g\theta(\Lambda) = diag(\theta)$. Whereas the Limitations of the Eigen decomposition cost is high for large graphs. The non-generalizable filters are graph-dependent.

B. *Spatial Graph Convolution*

In the Spatial domain the convolution is aggregated from local node's neighborhood and is expressed as

$$h_v^{(k+1)} = \sigma\left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{1}{C_{vu}} W^{(k)} h_u^{(k)}\right)$$

Here, $\boldsymbol{h_v^{(l)}}$ represents vector of node v at layer ***l***, ***N(v)*** is neighbors, $\boldsymbol{C_{vu}}$ is a normalization factor which is given by $\sqrt{\deg(v)\deg(u)}$ and $\boldsymbol{W^{(l)}}$ is the weight matrix. This method directly propagates and aggregates without leveraging any graph spectrum or Eigen vectors.

VI. Key Architectures in Graph Neural Networks(GNNs)

A. *Graph Convolutional Network (GCN): The basic Model*

Graph Convolution Network (GCNs) is an extended concept of convolution that is traditionally applied to structured data to irregular graph-structured data. Unlike the Euclidean convolutions that operate with a fixed receptive fields, GCNs support in the graph's adjacency structure to propagate and transform node features, where the idea is to aggregate information from a node's immediate neighbors while preserving the graph's properties. The layer-wise propagation rule used in my implementation is represented mathematically as:

$$H(l+1) = \sigma(D^{-1/2} A D^{-1/2} H^{(L)} W^{(l)}) \quad [8]$$

Here, $A = A + I$ [8]is the adjacency matrix, which ensures the nodes retain their features during aggregation. ***D*** is the degree matrix of *A*, [1] as shown in equation (3).

The nonlinear activation function is given by $\boldsymbol{\sigma}$ (e.g., ReLU), and the input feature matrix $\boldsymbol{H^{(0)} = X}$ represents the node features at layer 0. Each GCN layer aggregates with its next neighbor only, since it can lead to over-smoothing. The classic GCNs require full graph training, making them unfit for unseen graphs [1].

The GCNs have a learnable matrix where the weights are optimized during the phase of updating. Neighborhood hoping is also called stacking multiple GCN layers on top of each other as shown in figure 2. The output size varies according to the requirement, as shown below.

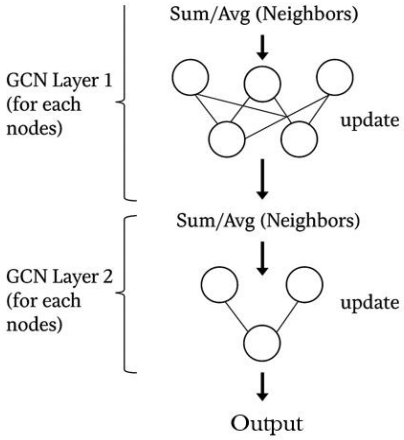


Fig 2 : Representation of a 2-layered GCN model

B. *Graph Attention Network (GAT): Attention Mechanism*

In GATs first, the state of all input node feature vectors is given by $\boldsymbol{(h_i)}$ to the GA-layer. For each node ***(i),*** the obtained feature vector is more structured and is more aware of its neighborhood. This is done by calculating a weighted sum of next node features, followed by a non-linear activation function σ.

The input layer is a set of nodes where N is the number of nodes and F is the number of features present in each node. The layer then produces a new set of node features [3], which is denoted by:

$$\mathbf{h}' = \{\vec{h}'_1, \vec{h}'_2, \ldots, \vec{h}'_N\}, \vec{h}'_i \in \mathbb{R}^{F'}$$

To get at least one learnable linear transformation, we need an input feature with expressive power to transform the input features to higher-level features. So a weight matrix is parameterized where it is applied at every node, and then the self–attention on nodes is shared, which indicates the importance of node j's features to node ***i***.

To make coefficients easily comparable across different nodes, we normalize them across all choices of j using the soft-max function [6]:

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

In GATs, each node aggregates messages using an attention mechanism to know the amount of influence each neighbor has, and their representations are updated based on the aggregated and weighted information. To increase the receptive field, the layers can be increased to two layers, which enables GNNs to capture more complex patterns and relationships between the graph data.

Offer a strong framework for learning from graph-structured data with multiple important advantages over other conventional deep learning architectures. The GNNs are unique as they incorporate both node features and the topological structure. Unlike models such as CNNs or RNNs which deal with grid or sequential data, GNNs are particularly well-suited to handle irregular, non-Euclidean spaces such as social networks, molecules, sensor networks, and knowledge graphs. This bestows special relevance to them for real-world scenarios where relational information is equally significant compared to node attributes.

C. Advantages of Graph Neural Networks

Graph Neural Network (GNNs) offer a strong framework for learning from graph-structured data with multiple advantages over conventional deep learning architectures. The GNNs are unique as they incorporate both node features and the topological structure. Unlike models such as CNNs or RNNs which deal with grid or sequential data, GNNs are particularly well-suited to handle irregular, non-Euclidean spaces such as social networks, molecules, sensor networks, and knowledge graphs. This bestows special relevance to them for real-world scenarios where relational information is equally significant compared to node attributes [1].

Another primary benefit of GNNs is enabling nodes to aggregate information from neighbors iteratively by using the message-passing mechanism. This implies that each node's representation develops over time based on both its features as well as the features of nodes in its multi-hop neighborhood, allowing context-aware representations.

It denotes that the model output is independent of the node and edge orderings within a graph. Graphs tend to be unordered with respect. The node IDs or orderings since these do not have any semantic meaning. GNNs also enable end-to-end differentiability, hence can be directly combined with other neural network components in sophisticated pipelines, e.g., vision-language models, recommendation systems, or reinforcement learning agents.

A practical reason GNNs are scalable is that they can be trained efficiently by applying stochastic techniques. For example, neighbor sampling in Graph SAGE allows it to be used for large graphs with nodes and edges numbering in the millions. The architectures are also flexible; for instance, GCNs, GATs, and GINs, are designed to cater to different needs of application domains as well as theoretical needs, such as attention mechanism, maximal expressiveness, and inductive learning [7].

D. Challenges and Limitations of Graph Neural Networks

One of the basic limitations arises from the expressive power of classical GNN architectures like GCN and GraphSAGE due to the inherent constraints of neighborhood aggregation [4]. The Weisfeiler-Lehman (WL) test for graph isomorphism limits these models by suggesting that different nodes or graphs with identical neighborhood structures can result in identical embeddings, which lowers the discriminative power of the model [4].

Over-smoothing occurs when the information is repeatedly aggregated from neighbours across multiple GNN layers, which presents another task [2]. Node representations tend to become indefinite as network depth increases, making it challenging to maintain significant differences, particularly in large or deep networks. This restricts how deep GNNs can be stacked efficiently and frequently calls for regularization methods or architectural changes to mitigate.

A related issue is over-squashing, in which too much information is squashed into fixed-size representations for nodes, especially when the mechanism involves long-distance message passing. The model fairs worse on tasks that make use of global context when information from distant nodes has to be forced through narrow bottlenecks (such as shallow embeddings or limited depth), which causes the model to lose crucial structural signals.

The computational complexity of GNNs is increased by their frequent poor scalability with respect to graph size. Full-batch training requires loading the entire adjacency matrix and feature set into memory, which is not possible in large-scale graphs like web networks or industrial knowledge graphs. Sampling-based methods like GraphSAGE help with this to some extent, but they also add variance and approximation error [4].

Noise and sparse data is other factors that very likely might have an influence on GNN performance [3]. GNNs rely heavily on the connectivity structure of correctness; thus, missing edges or wrong edges, as can occur in real-world data, may confound the model at the training stage. Again, if it is sparse, effectiveness with regards to message passing might be limited with nodes having fewer connections due to inadequate contextual information.

E. Applications of Graph Neural Network(GNNs)

By utilizing both node features and the underlying graph structure, Graph Neural Networks (GNNs) have proven to be remarkably versatile in a variety of domains. The main application domains where GNNs have demonstrated a noteworthy influence are listed below.

1) Social Networks(Friend Recommendation): In social networks such as Facebook or LinkedIn, one can naturally model users and their interactions as a graph with nodes representing the users and connections representing friendship or an interaction between them. GNNs are powerful in learning user representations by aggregating neighbors' information in networks about particular users. Friend recommendation is prominent, where the GNN analyzes the structure of the network as well as the attributes of the users to point out those with whom one might create a new connection: for example, GNNs can return those people who might be mutual friends to you or who have similar interests. The other important application of GNNs is influencer identification, wherein users get classified as either an influencer or a non-influencer on the basis of direct and indirect connections. GNNs shine in these kinds of problems as they naturally treat the relational aspects of social networks and scale elegantly to very large graphs while keeping the notion of context in terms of each user's position in the network.

2) Chemistry(Drug Discovery): The GNNs work toward molecular classification and property prediction, treating the molecule as a single entity. Such a graph captures interdependencies among atoms which would otherwise be essential to biochemical predictions dependent on molecular

structure. Thus, the power of modelling such intricate relationships both provides an aid to GNNs for drug discovery ad chemical analysis.

3) Fraud Detection and Recommendation Systems: In financial systems and online platforms, certain entities like users, transactions, and devices may get modeled as graphs. In fraud detection, GNNs are very well employed to spot suspicious behaviours by pointing the anomalous subgraphs or instance interaction patterns, including the case of a coalesced fraud ring. For recommendation systems, GNNs work on user-item interaction graphs to model direct as well as indirect affinities. While considering items that a user has interacted with, it also considers the structure of those interactions. It thus offers a big difference in recommendation quality. Being powerful in both local and higher-order structures, they are well equipped to detect subtle fraudulent activities and nuanced user preferences.

4) Automotive Applications: In the automotive industry, GNNs are increasingly applied to intelligent transportation systems by modelling road networks or vehicle-to-vehicle communication graphs, where nodes represent vehicles or intersections and edges represent physical or wireless connections. GNNs analyze traffic flow, predict congestion, and optimize routing by analyzing and learning about vehicles involved in GNNs while handling data from sensors connected, like lidar, radar, and cameras and provide a unified interpretation of the environment across all sensor modalities.

5) Computer Vision: In computer vision, where images can be represented as graphs of connected regions or super pixels, GNNs have been used. Here, the edges depict the spatial relationships between homogeneous regions and nodes to represent the spatial interactions between them. Relational reasoning helps GNNs perform better on challenging visual recognition tasks by helping them understand the spatial hierarchy and relationships between objects.

6) Cyber Security(Network Intrusion Detection): Network data in cyber security can be denoted as a graph, where nodes stand for devices and edges for the connections between them. Some techniques Malware analysis, network interruption detection, and network traffic anomaly detection are all tasks performed by GNNs. Complicated or secret attacks that don't follow regular patterns are often difficult for conventional strategies to identify. Learning from these relational and structural features of the network, GNNs can identify odd patterns and slight indications of compromise.

VII. Implementation of Graph Neural Network for ECU fault detection

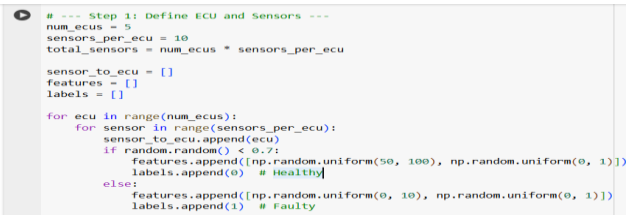
For implementation of this example, I have used the same mathematical concepts and equations that was discussed in the topics above. I have used the Python programming language and Pytorch framework for the implementation of this example. I have also used Google colab as a notebook environment.

The Pytorch Geometric assigns an initial node embedding's using an attribute of the data object. During each iteration of message passing, this framework uses Message passing class to define the message passing operation and aggregate messages from neighboring nodes. This information is combined with the present embedding of the node and the iterations of message passing continue for two layers and the resulting nodes embedding's can be used for various downstream tasks as shown for node classification.

In Automotive systems, Electronic circuit units (ECUs) are responsible for receiving real-time data from multiple sensors. The health and reliability of each sensor is critical for the safe operation of any vehicle. Traditional diagnostic methods frequently ignore the relational structure that exists between sensors both within and across ECUs, and focus on analyzing individual sensors.

To address this limitation, I have implemented a sample code that shows a node classification task on a graph-structured representation of the sensor network. According to its characteristics and the combined data from nearby nodes, each node is to be categorized as either healthy (label 0) or defective (label 1). Now, the following are some code snippets and results from my implementation.

A. Data Preprocessing



```
# --- Step 1: Define ECU and Sensors ---
num_ecus = 5
sensors_per_ecu = 10
total_sensors = num_ecus * sensors_per_ecu

sensor_to_ecu = []
features = []
labels = []

for ecu in range(num_ecus):
    for sensor in range(sensors_per_ecu):
        sensor_to_ecu.append(ecu)
        if random.random() < 0.7:
            features.append([np.random.uniform(50, 100), np.random.uniform(0, 1)])
            labels.append(0)  # Healthy
        else:
            features.append([np.random.uniform(0, 10), np.random.uniform(0, 1)])
            labels.append(1)  # Faulty
```

Fig 3: Data preparation

The figure shows a piece of code that defines the ECU and sensors. The simulation starts with defining 5 ECUs, with each ECU having 10 sensors, providing 50 sensors in the network totally. The code used the sensor_to_ecu mapping, which assigned each sensor to its respective ECU and created two features. The first feature simulates the sensor reading (e.g., temperature), where healthy sensors (70% chance) are assigned values between 50 and 100, while faulty sensors (30% chance) are given values between 0 and 10 to mimic abnormal behavior. An auxiliary reading normalized between 0 and 1 is the second feature. Moreover, labels are made, where 0 denotes a healthy sensor and 1 denotes a faulty one. For training a GNN, the structured approach is considered, where the model must learn to differentiate between normal and anomalous sensor readings based on their feature distributions and graph connections. The randomness (random.random()) simulates real-world sensor data imperfections by introducing variability. The groundwork for the graph construction and GNN training process, the initialization phase, is critical.

B. GCN model

```
# --- Step 3: GCN Model ---
class GCN(torch.nn.Module):
    def __init__(self):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(2, 16)
        self.conv2 = GCNConv(16, 2)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        self.embeddings = x.detach()
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

Fig 4: GCN Model

The Graph Convolution Network class has been implemented as a PyTorch module with two components: a forward propagation and initialization (__init__) methods.in the initialization two graph convolution layers are defined: Two input features(corresponding to the sensor reading) are converted into a 16- dimensional hidden representation by the first layer (conv1), and this 16- dimensional space is then reduced into two output dimensions for binary classification (healthy vs. faulty) by the second layer(conv2).

The forward method extracts the node features (x) and edge connections (edge_index) of the input data, then calls the first convolutional layer, and finally a ReLU activation function to introduce non-linearity. Following activation, the second convolutional layer uses these features, and the outputs are normalized using log-softmax to log probabilities for each class.

In addition to preserving the graph structure, this architecture enables the network to efficiently aggregate and transform input from nodes that are closer to one another. The dimensionality reduction from 16 to 2 features helps in the extraction of the most relevant data for the classification task. In addition to removing the problem of gradient disappearance during back propagation, the ReLU activation across layers guarantees that the model will learn complex, nonlinear correlations in the sensor input.

C. Training Loop

```
# --- Step 6: Training ---
epochs = 200
loss_values = []

for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    out = model(data)
    loss = F.nll_loss(out, data.y)
    loss.backward()
    optimizer.step()

    loss_values.append(loss.item())

    if epoch % 10 == 0:  # Print every 10 epochs
        print(f"Epoch: {epoch}  loss: {loss.item()}")
```

Fig 5: Training loop

The graph's weights are updated during the training process using the Adam optimizer, which has a learning rate of 0.01 over 200 epochs. This loss is suitable for classification since it improves the log-softmax output of the model and produces incorrect predictions.

During each training epoch, the model tundergoes several important steps. First, the model is set to training mode to ensure that all layers work correctly. The forward pass first involves obtaining the model's predictions by passing the input features through both GCN layers. The input is the 2-dimensional sensor data, which is transformed originally into the 16-dimensional latent space before being projected to the final 2-class output.

The NLL Loss is then calculated by comparing predictions to ground truth tables, which gives a value that shows the current performance of the model. The Adam Optimizer can then take these gradients and apply them to the model's weights, making updates in the direction to minimize the loss function. Adam is very useful to this process because its adaptive learning rate can scale the updates for each parameter as it depends on the historic gradients.

To track the training process, the loss value is recorded each epoch and output every 10 iterations. This allows the user to observe the model as it learns and determine if optimization has occurred by noting a steady decline in loss. The loss values were saved in a list for analysis after training, allowing learning curves to be created, which could indicate problems like over-fitting or insufficient training. With these three elements—a suitable optimizer, a suitable loss function and a considered training loop—the GNN model can learn to distinguish a healthy sensor from faulty sensor while achieving stability during the optimization process.

D. Results and Analysis

The training loss curve (Figure) shows consistent improvement across 200 epochs, with overall loss decreasing from 0.382 to 0.372. The steady decline indicates that back propagation optimization of GNNs parameters was successful. The loss reduction occurs in three phases: (1) Rapid improvement in early epoch (0-50) when model is learning base patterns in the data, (2) moderate improvement with more refinement of patterns in the middle epochs (50-150) as the model is adjusting the finer details of the patterns, and (3) near convergence in the later epochs (150-200) as the curve flattens indicating that the model has extracted most of the learnt patterns from the training data. The final loss of 0.372 provides an indicator of stable performance, suggesting the GNN was able to best differentiate between healthy and faulty sensors by effectively passing messages in the iteration across the graph structure.

In addition to the graphical representation, to track the training progress numerically, the loss at every 10-epoch interval was printed as shown in the figure. The figure shows the loss values observed when the training was undergoing. The gradual decrease in the loss value indicates that the model training was successful.

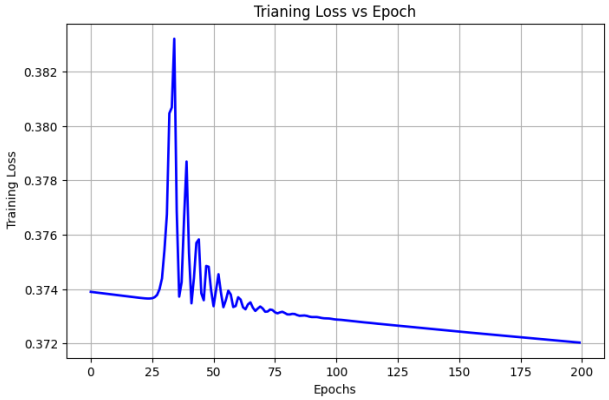


Fig 6 : Training loss vs Epoch Graphical representation

we plotted the sensor network at different stages of training as shown in Figure 6.

```
Epoch-wise Loss:

Epoch 0: Loss = 3.773800
Epoch 10: Loss = 0.944501
Epoch 20: Loss = 0.455182
Epoch 30: Loss = 0.514556
Epoch 40: Loss = 0.449488
Epoch 50: Loss = 0.429326
Epoch 60: Loss = 0.411770
Epoch 70: Loss = 0.391401
Epoch 80: Loss = 0.387409
Epoch 90: Loss = 0.386339
Epoch 100: Loss = 0.385032
Epoch 110: Loss = 0.383972
Epoch 120: Loss = 0.383036
Epoch 130: Loss = 0.382254
Epoch 140: Loss = 0.381578
Epoch 150: Loss = 0.381013
Epoch 160: Loss = 0.380545
Epoch 170: Loss = 0.380161
Epoch 180: Loss = 0.379846
Epoch 190: Loss = 0.379584
```

Fig 7: Epoch loss

The initial state, where sensors are colored according to their corresponding ECUs, is depicted in Fig. 1. To improve differentiation, each ECU has a unique color (such as coral, green, blue, pink, or yellow). No health classification is known at this time. Figures 8, 9, and 10 illustrate the intermediate state of a few training sessions, such as Epoch 0, Epoch 150, Epoch 100, and Epoch 150. In the figure, some sensor nodes turn red (faulty) or green (healthy) based on the GNNs' evolving predictions. However, not all nodes are correctly classified at this stage and indicating an ongoing learning process.

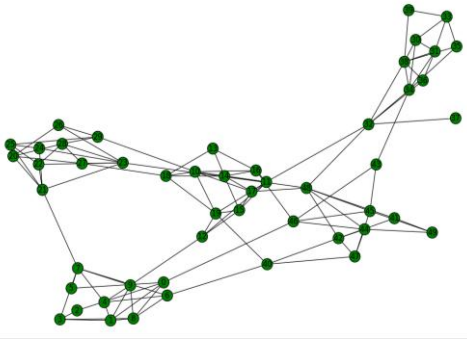


Fig 8: Sensor network Node classification during Training Epoch 0

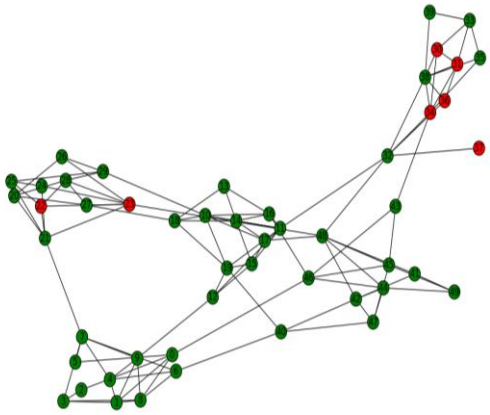


Fig 9: Sensor network Node classification during Training Epoch 50

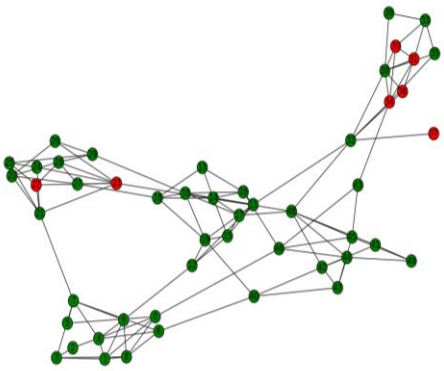


Fig 10 : Sensor network Node classification during Training Epoch 100

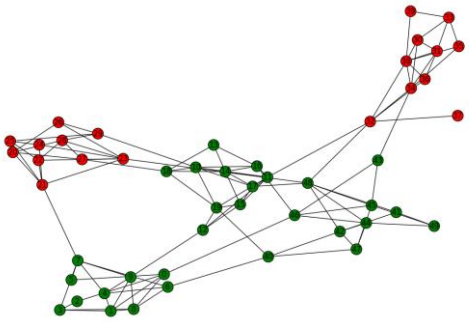


Fig11: Node classification displaying the faulty and healthy sensors

Figure 11 displays the final classification after the completion of training, along with the ECU health status as

shown in Figure 12. The GNN successfully identifies healthy and faulty sensors. Nodes belonging to faulty ECUs are marked in red and the healthy ones in green, enabling clear fault detection at the ECU level. This higher level aggregation will help ease the risk that, even though a single critical sensor could fail, it will effectively provide a means to classify the entire ECU for inspection as the ECU was dependent on the faulty sensor, which is similar to what happens in real-world automotive diagnostics, as a faulty sensor can affect the function of the ECU as a whole.

```
ECU Health Status (After Training):

ECU 1: True = Faulty, Predicted = Healthy
ECU 2: True = Faulty, Predicted = Healthy
ECU 3: True = Faulty, Predicted = Faulty
ECU 4: True = Faulty, Predicted = Faulty
ECU 5: True = Faulty, Predicted = Healthy
```

Figure 11: ECU health status (After training)

This result highlights the GNN's ability to accurately identify single nodes (sensors) and conclude the health status of ECUs. This hierarchical fault detection is crucial in automotive systems, as finding faulty ECUs at an early stage can prevent large system failures, ensuring safety and reliability.

VIII. Conclusion

In Conclusion, examining both theoretical and real-world applications of Graph Neural Networks, the fundamental concepts of message passing and graph convolution operations with notations and pointing the important properties like permutation invariance and local neighborhood aggregation between well-known architectures (GCN and GAT).To ground these concepts, I proposed a proof-of-concept(implementation) - two-layer Graph Convolutional Network (GCN) applied to a synthetic sensor(node) network where each sensor's simulated readings were leveraged to learn node embedding's and classify health status of an ECUs(node classification) in an automotive system. Aggregating sensor-level predictions demonstrated how GNNs can effectively recognize ECU-level faults when any connected sensor is defective.

Henceforth, my implementation mainly focuses on the feasibility of using GNNs for structured data. Further studies will aim to mitigate the limitations like over-smoothing, over-squashing through hierarchical pooling and incorporate real-world automotive data sets, expand the model to include dynamic graph updates, scale these approaches to larger systems, and resolve well-known issues.

Acknowledgment

I would like to thank Prof. Dr.A. Pech and I. Pole for allowing me to work on this topic and for giving me the proper guidance to write this paper .I would specially like to thank I. Pole for his insightful explanations and engaging discussions during the sessions, which greatly enhanced my understanding if the topic

Declaration

I hereby declare that I have completed the presented seminar paper independently, without using anything other than the specified literature and aids. All parts taken from published and non-published texts, verbally or in substance, are marked as such. This report has not been to any examination office in this way.

Signature:
Date: 14/06/2025

References

[1] Khemani, B. et al. (2024). A review of graph neural networks: concepts, architectures, techniques, challenges, datasets, applications, and future directions. Journal of Big Data, 11(1). https://doi.org/10.1186/s40537-023-00876-4.

[2] J. Bhadra, A. S. Khanna, and A. B. H., "A Graph Neural Network Approach for Identification of Influencers and Micro-Influencers in a Social Network: Classifying Influencers from Non-Influencers using GNN and GCN," in Proc. Int. Conf. on Advances in Electronics, Communication, Computing and Intelligent Information Systems (ICAECIS), 2023, pp. 979–983, doi: 10.1109/ICAEIS58353.2023.10170708.

[3] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "Computational Capabilities of Graph Neural Networks," IEEE Trans. Neural Netw., vol. 20, no. 1, pp. 81–102, Jan. 2009, doi: 10.1109/TNN.2008.2005141.

[4] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are Graph Neural Networks?" in Proc. Int. Conf. Learn. Representations (ICLR), 2019. [Online]. Available: https://arxiv.org/abs/1810.00826.

[5] M. Tiezzi, G. Ciravegna, and M. Gori, "Graph Neural Networks for Graph Drawing," IEEE Trans. Neural Netw. Learn. Syst., vol. 35, no. 4, pp. 4668–4683, Apr. 2024, doi: 10.1109/TNNLS.2022.318496

[6] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 44, no. 1, pp. 202–215, 2022. [DOI: 10.1109/TPAMI.2021.3073379].

[7] P. Gupta and P. K. Gupta, "Performance Analysis of GCN, GNN, and GAT Models with Differentiable Pooling for Detection of Fake News," in Proc. 3rd IEEE Delhi Section Flagship Conference (DELCON), 2024, pp. 987–992, doi: 10.1109/DELCON64804.2024.10866089.

[8] Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," IEEE Transactions on Neural Networks and Learning Systems, vol. 32, no. 1, pp. 4–24, Jan. 2021, doi: 10.1109/TNNLS.2020.2978386.

[9] A. K. Awasthi, A. K. Garov, M. Sharma, and M. Sinha, "GNN Model Based on Node Classification Forecasting in Social Network," in Proc. Int. Conf. on Artificial Intelligence and Smart Communication (AISC), 2023, pp. 1039–1044, doi: 10.1109/AISC56616.2023.10085118.