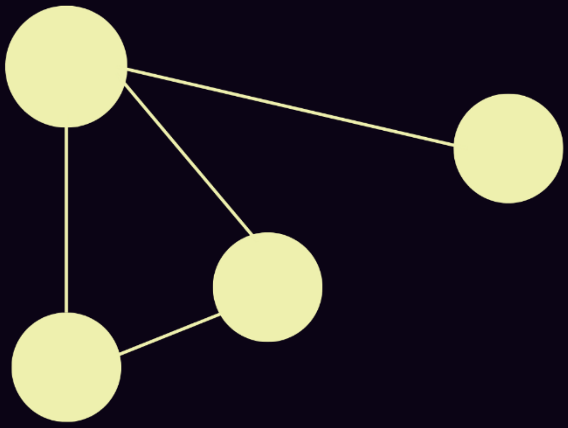
Graph Neural Networks



2

From Theory to Practice A Deep Dive into Implementation and Applications

Era

Graph Neural Networks: From Theory to Practice A Deep Dive into Implementation and Applications (Part 2)

Graph Neural Networks (GNNs) represent a significant advancement in machine learning's ability to process and analyze graph-structured data. This document presents a comprehensive analysis of Graph Neural Networks' implementation and practical applications, building upon the theoretical framework established in Part 1. The analysis examines the architectural components and learning mechanisms that enable GNNs to process graph-structured data effectively. Particular attention is given to the state update function, output transformation processes, and the sophisticated learning algorithms that allow GNNs to adapt to various graph structures. Through detailed examination of both linear and non-linear implementation approaches, this paper bridges the gap between theoretical principles and practical applications, demonstrating how GNNs have evolved from a theoretical framework to power modern applications in areas such as social network analysis, molecular structure prediction, and recommendation systems.

Graph Neural Networks: From Theory to Practice A Deep Dive into Implementation and Applications (Part 2)

Author Analysis & Documentation

By: Hussein Mahdi Fakhry

Master's Student in Software Development

University of Babylon / College of Information Technology

Document Version: 1.0

Published: Feb 3,2025

Last Updated: Feb 4,2025

Documentation Notice:

This document represents an original analysis of the 2009 research paper "The Graph Neural Network Model" by Franco Scarselli et al. While discussing and analyzing the original paper's content, this analysis contains original insights, explanations, and interpretations that are the intellectual property of the author.

Citation Requirements:

For academic or professional reference, please cite this work as:

Fakhry, H. M. (2025). "Graph Neural Networks: From Theory to Practice A Deep Dive into Implementation and Applications (Part 2)". Published on Medium and GitHub.

Contact Information:

GitHub: https://github.com/Hu8MA

LinkedIn: https://www.linkedin.com/in/hussein16mahdi/

Professional Email: hussein.mahdifa@gmail.com

Graph Neural Networks: From Theory to Practice A Deep Dive into Implementation and Applications (Part 2)

The Graph Neural Network Model

Franco Scarselli, Marco Gori, *Fellow, IEEE*, Ah Chung Tsoi, Markus Hagenbuchner, *Member, IEEE*, and Gabriele Monfardini

***Abstract*—Many underlying relationships among data in several areas of science and engineering, e.g., computer vision, molecular chemistry, molecular biology, pattern recognition, and data mining, can be represented in terms of graphs. In this paper, we propose a new neural network model, called graph neural network (GNN) model, that extends existing neural network methods for processing the data represented in graph domains. This GNN model, which can directly process most of the practically useful types of graphs, e.g., acyclic, cyclic, directed, and undirected, implements a function $\tau(\boldsymbol{G}, n) \in \mathbb{R}^m$ that maps a graph $\boldsymbol{G}$ and one of its nodes n into an m-dimensional Euclidean space. A supervised learning algorithm is derived to estimate the parameters of the proposed GNN model. The computational cost of the proposed algorithm is also considered. Some experimental results are shown to validate the proposed learning algorithm, and to demonstrate its generalization capabilities.**

***Index Terms*—Graphical domains, graph neural networks (GNNs), graph processing, recursive neural networks.**

I. Introduction

DATA can be naturally represented by graph structures in several application areas, including proteomics [1], image analysis [2], scene description [3], [4], software engineering [5], [6], and natural language processing [7]. The simplest kinds of graph structures include single nodes and sequences. But in several applications, the information is organized in more complex graph structures such as trees, acyclic graphs, or cyclic graphs. Traditionally, data relationships exploitation has been the subject of many studies in the community of inductive logic programming and, recently, this research theme has been evolving in different directions [8], also because of the applications of relevant concepts in statistics and neural networks to such areas (see, for example, the recent workshops [9]–[12]).

In machine learning, structured data is often associated with the goal of (supervised or unsupervised) learning from examples a function τ that maps a graph G and one of its nodes n to a vector of reals[1]: $\tau(\boldsymbol{G}, n) \in \mathbb{R}^m$. Applications to a graphical domain can generally be divided into two broad classes, called *graph-focused* and *node-focused* applications, respectively, in this paper. In *graph-focused* applications, the function τ is independent of the node n and implements a classifier or a regressor on a graph structured data set. For example, a chemical compound can be modeled by a graph $\boldsymbol{G}$, the nodes of which stand for atoms (or chemical groups) and the edges of which represent chemical bonds [see Fig. 1(a)] linking together some of the atoms. The mapping $\tau(\boldsymbol{G})$ may be used to estimate the probability that the chemical compound causes a certain disease [13]. In Fig. 1(b), an image is represented by a region adjacency graph where nodes denote homogeneous regions of intensity of the image and arcs represent their adjacency relationship [14]. In this case, $\tau(\boldsymbol{G})$ may be used to classify the image into different classes according to its contents, e.g., castles, cars, people, and so on.

In *node-focused* applications, τ depends on the node n, so that the classification (or the regression) depends on the properties of each node. Object detection is an example of this class of applications. It consists of finding whether an image contains a given object, and, if so, localizing its position [15]. This problem can be solved by a function τ, which classifies the nodes of the region adjacency graph according to whether the corresponding region belongs to the object. For example, the output of τ for Fig. 1(b) might be 1 for black nodes, which correspond to the castle, and 0 otherwise. Another example comes from web page classification. The web can be represented by a graph where nodes stand for pages and edges represent the hyperlinks between them [Fig. 1(c)]. The web connectivity can be exploited, along with page contents, for several purposes, e.g., classifying the pages into a set of topics.

Traditional machine learning applications cope with graph structured data by using a preprocessing phase which maps the graph structured information to a simpler representation, e.g., vectors of reals [16]. In other words, the preprocessing step first "squashes" the graph structured data into a vector of reals and then deals with the preprocessed data using a list-based data processing technique. However, important information, e.g., the topological dependency of information on each node may be lost during the preprocessing stage and the final result may depend, in an unpredictable manner, on the details of the preprocessing algorithm. More recently, there have been various approaches [17], [18] attempting to preserve the graph structured nature of the data for as long as required before the processing

Manuscript received May 24, 2007; revised January 08, 2008 and May 02, 2008; accepted June 15, 2008. First published December 09, 2008; current version published January 05, 2009. This work was supported by the Australian Research Council in the form of an International Research Exchange scheme which facilitated the visit by F. Scarselli to University of Wollongong when the initial work on this paper was performed. This work was also supported by the ARC Linkage International Grant LX045446 and the ARC Discovery Project Grant DP0453089.

F. Scarselli, M. Gori, and G. Monfardini are with the Faculty of Information Engineering, University of Siena, Siena 53100, Italy (e-mail: franco@dii.unisi.it; marco@dii.unisi.it; monfardini@dii.unisi.it).

A. C. Tsoi is with Hong Kong Baptist University, Kowloon, Hong Kong (e-mail: act@hkbu.edu.hk).

M. Hagenbuchner is with the University of Wollongong, Wollongong, N.S.W. 2522, Australia (e-mail: markus@uow.edu.au).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNN.2008.2005605

[1]Note that in most classification problems, the mapping is to a vector of integers $\mathbb{N}^m$, while in regression problems, the mapping is to a vector of reals $\mathbb{R}^m$. Here, for simplicity of exposition, we will denote only the regression case. The proposed formulation can be trivially rewritten for the situation of classification.

In Part 1, we explored the theoretical foundations that make Graph Neural Networks (GNNs) such a revolutionary approach to processing graph-structured data. We uncovered how the tau function, information diffusion, and universal approximation principles come together to create a powerful framework for understanding complex relationships in data.

Now, let's roll up our sleeves and dive into the practical aspects that make GNNs work in the real world. We'll explore the intricate architecture that brings these theoretical concepts to life, unpack the learning algorithms that allow GNNs to adapt and improve, and examine different implementation approaches that address various real-world challenges. Along the way, we'll see how researchers transformed elegant mathematical principles into practical solutions that are reshaping how we handle interconnected data.

Understanding GNN's Key Functions: A Closer Look at the Core Mechanisms

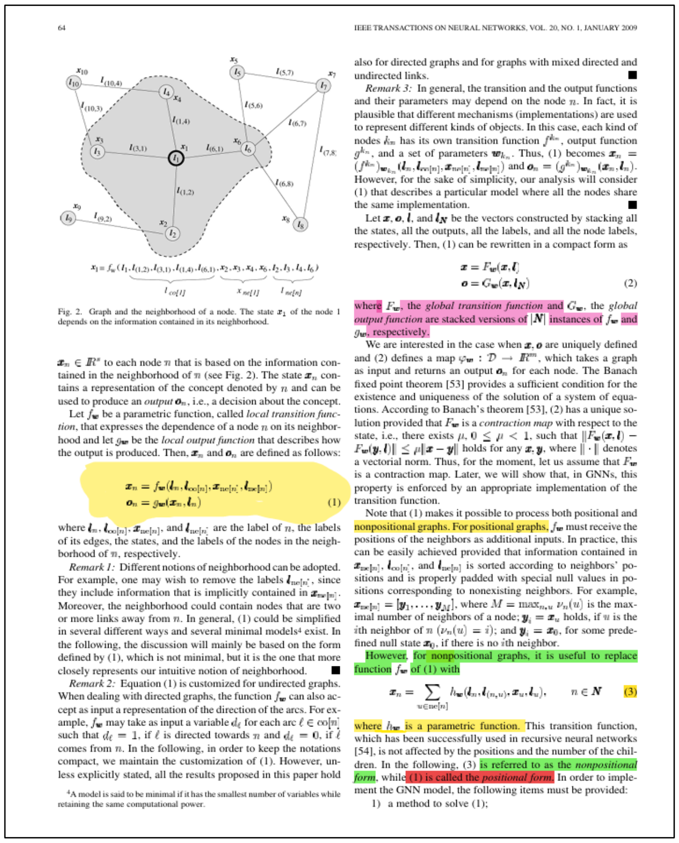
When examining pages 64-65 of the paper, we encounter two fundamental functions that form the backbone of GNN operations, along with a crucial mechanism that ensures their proper coordination over time. Let's break them down in a way that reveals their practical significance, using a real-world example to illuminate their roles.

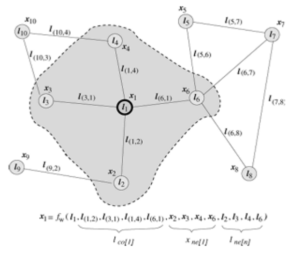
The State Update Function: The Network's Information Processor

1. State Update Function (fω)

- Node's own features (l_n)
- Edge features (lco[n])
- Neighbor states (Xne[n])
- Neighbor features (lne[n])

*it a simple form equation is called **"positional form"**





$$x_1 = f_w(l_1, \underbrace{l_{(1,2)}, l_{(3,1)}, l_{(1,4)}, l_{(6,1)}}_{l_{co[1]}}, \underbrace{x_2, x_3, x_4, x_6}_{x_{ne[1]}}, \underbrace{l_2, l_3, l_4, l_6}_{l_{ne[n]}})$$

Fig. 2. Graph and the neighborhood of a node. The state $\boldsymbol{x}_1$ of the node 1 depends on the information contained in its neighborhood.

$\boldsymbol{x}_n \in \mathbb{R}^s$ to each node n that is based on the information contained in the neighborhood of n (see Fig. 2). The state $\boldsymbol{x}_n$ contains a representation of the concept denoted by n and can be used to produce an *output* $\boldsymbol{o}_n$, i.e., a decision about the concept.

Let $f_{\boldsymbol{w}}$ be a parametric function, called *local transition function*, that expresses the dependence of a node n on its neighborhood and let $g_{\boldsymbol{w}}$ be the *local output function* that describes how the output is produced. Then, $\boldsymbol{x}_n$ and $\boldsymbol{o}_n$ are defined as follows:

$$\begin{aligned} \boldsymbol{x}_n &= f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{\text{co}[n]}, \boldsymbol{x}_{\text{ne}[n]}, \boldsymbol{l}_{\text{ne}[n]}) \\ \boldsymbol{o}_n &= g_{\boldsymbol{w}}(\boldsymbol{x}_n, \boldsymbol{l}_n) \end{aligned} \qquad (1)$$

where $\boldsymbol{l}_n$, $\boldsymbol{l}_{\text{co}[n]}$, $\boldsymbol{x}_{\text{ne}[n]}$, and $\boldsymbol{l}_{\text{ne}[n]}$ are the label of n, the labels of its edges, the states, and the labels of the nodes in the neighborhood of n, respectively.

Remark 1: Different notions of neighborhood can be adopted. For example, one may wish to remove the labels $\boldsymbol{l}_{\text{ne}[n]}$, since they include information that is implicitly contained in $\boldsymbol{x}_{\text{ne}[n]}$. Moreover, the neighborhood could contain nodes that are two or more links away from n. In general, (1) could be simplified in several different ways and several minimal models[4] exist. In the following, the discussion will mainly be based on the form defined by (1), which is not minimal, but it is the one that more closely represents our intuitive notion of neighborhood. ■

Remark 2: Equation (1) is customized for undirected graphs. When dealing with directed graphs, the function $f_{\boldsymbol{w}}$ can also accept as input a representation of the direction of the arcs. For example, $f_{\boldsymbol{w}}$ may take as input a variable d_ℓ for each arc $\ell \in \text{co}[n]$ such that $d_\ell = 1$, if ℓ is directed towards n and $d_\ell = 0$, if ℓ comes from n. In the following, in order to keep the notations compact, we maintain the customization of (1). However, unless explicitly stated, all the results proposed in this paper hold also for directed graphs and for graphs with mixed directed and undirected links. ■

Remark 3: In general, the transition and the output functions and their parameters may depend on the node n. In fact, it is plausible that different mechanisms (implementations) are used to represent different kinds of objects. In this case, each kind of nodes k_n has its own transition function f^{k_n}, output function g^{k_n}, and a set of parameters $\boldsymbol{w}_{k_n}$. Thus, (1) becomes $\boldsymbol{x}_n = (f^{k_n})_{\boldsymbol{w}_{k_n}}(\boldsymbol{l}_n, \boldsymbol{l}_{\text{co}[n]}, \boldsymbol{x}_{\text{ne}[n]}, \boldsymbol{l}_{\text{ne}[n]})$ and $\boldsymbol{o}_n = (g^{k_n})_{\boldsymbol{w}_{k_n}}(\boldsymbol{x}_n, \boldsymbol{l}_n)$. However, for the sake of simplicity, our analysis will consider (1) that describes a particular model where all the nodes share the same implementation. ■

Let $\boldsymbol{x}$, $\boldsymbol{o}$, $\boldsymbol{l}$, and $\boldsymbol{l}_{\boldsymbol{N}}$ be the vectors constructed by stacking all the states, all the outputs, all the labels, and all the node labels, respectively. Then, (1) can be rewritten in a compact form as

$$\begin{aligned} \boldsymbol{x} &= F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \\ \boldsymbol{o} &= G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) \end{aligned} \qquad (2)$$

where $F_{\boldsymbol{w}}$, the *global transition function* and $G_{\boldsymbol{w}}$, the *global output function* are stacked versions of $|\boldsymbol{N}|$ instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, respectively.

We are interested in the case when $\boldsymbol{x}, \boldsymbol{o}$ are uniquely defined and (2) defines a map $\varphi_{\boldsymbol{w}} : \mathcal{D} \to \mathbb{R}^m$, which takes a graph as input and returns an output $\boldsymbol{o}_n$ for each node. The Banach fixed point theorem [53] provides a sufficient condition for the existence and uniqueness of the solution of a system of equations. According to Banach's theorem [53], (2) has a unique solution provided that $F_{\boldsymbol{w}}$ is a *contraction map* with respect to the state, i.e., there exists μ, $0 \le \mu < 1$, such that $\|F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) - F_{\boldsymbol{w}}(\boldsymbol{y}, \boldsymbol{l})\| \le \mu \|\boldsymbol{x} - \boldsymbol{y}\|$ holds for any $\boldsymbol{x}, \boldsymbol{y}$, where $\|\cdot\|$ denotes a vectorial norm. Thus, for the moment, let us assume that $F_{\boldsymbol{w}}$ is a contraction map. Later, we will show that, in GNNs, this property is enforced by an appropriate implementation of the transition function.

Note that (1) makes it possible to process both positional and nonpositional graphs. For positional graphs, $f_{\boldsymbol{w}}$ must receive the positions of the neighbors as additional inputs. In practice, this can be easily achieved provided that information contained in $\boldsymbol{x}_{\text{ne}[n]}$, $\boldsymbol{l}_{\text{co}[n]}$, and $\boldsymbol{l}_{\text{ne}[n]}$ is sorted according to neighbors' positions and is properly padded with special null values in positions corresponding to nonexisting neighbors. For example, $\boldsymbol{x}_{\text{ne}[n]} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_M]$, where $M = \max_{n,u} \nu_n(u)$ is the maximal number of neighbors of a node; $\boldsymbol{y}_i = \boldsymbol{x}_u$ holds, if u is the ith neighbor of n ($\nu_n(u) = i$); and $\boldsymbol{y}_i = \boldsymbol{x}_0$, for some predefined null state $\boldsymbol{x}_0$, if there is no ith neighbor.

However, for nonpositional graphs, it is useful to replace function $f_{\boldsymbol{w}}$ of (1) with

$$\boldsymbol{x}_n = \sum_{u \in \text{ne}[n]} h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u), \qquad n \in \boldsymbol{N} \qquad (3)$$

where $h_{\boldsymbol{w}}$ is a parametric function. This transition function, which has been successfully used in recursive neural networks [54], is not affected by the positions and the number of the children. In the following, (3) is referred to as the *nonpositional form*, while (1) is called the *positional form*. In order to implement the GNN model, the following items must be provided:

1) a method to solve (1);

[4]A model is said to be minimal if it has the smallest number of variables while retaining the same computational power.

Imagine a social network where each person represents a node in our graph. The State Update Function (fω) acts like a sophisticated information gatherer, collecting and processing data from multiple sources. When updating a user's profile (let's call her Alice), this function considers four crucial pieces of information:

First, it looks at Alice's own characteristics - her interests, activity level, and profile data. Then, it examines her connections - how long she's been friends with others and the nature of these relationships. The function also considers her friends' current status in the network and their characteristics. All this information comes together to update Alice's "state" in the network.

The Output Function: Transforming States into Meaningful Results

2. Output Function (gω)

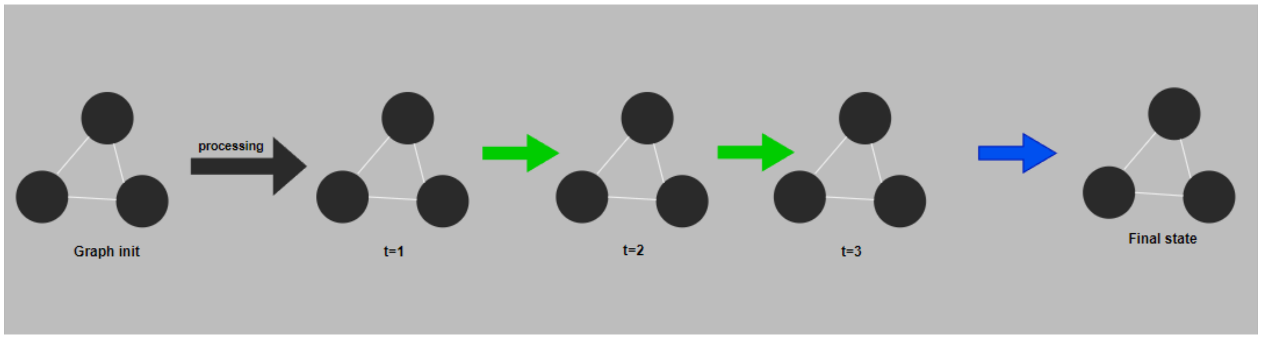
- Node state (x_n)
- Node features (l_n)

The Output Function (gω) represents one of the researchers' most elegant solutions. Rather than using simple mathematical operations, they implemented it as a multilayered feedforward neural network. This function takes Alice's updated state (all the processed information about her position and relationships in the network) along with her original features and transforms them into meaningful predictions - perhaps determining if she's likely to be an influential user or interested in certain content.

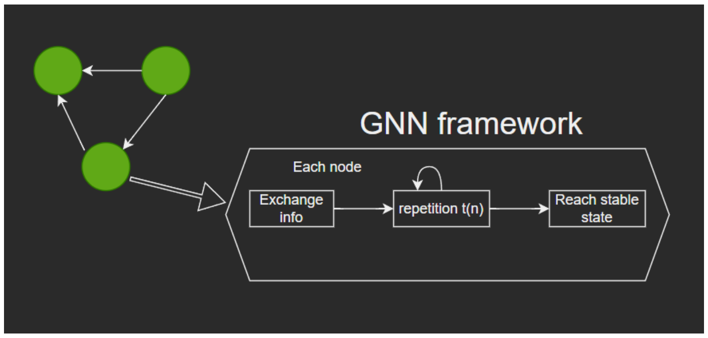
The Global Update: Orchestrating the Network's Evolution

$$\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l}) \quad (4)$$

What makes this framework particularly powerful is how these functions work together over time through the Global Update mechanism (Fw). Remember how in Part 1 we discussed GNNs' relationship with recursive neural networks? This is where that temporal aspect comes into play. The network doesn't just process information once; it continues to update and refine its understanding.



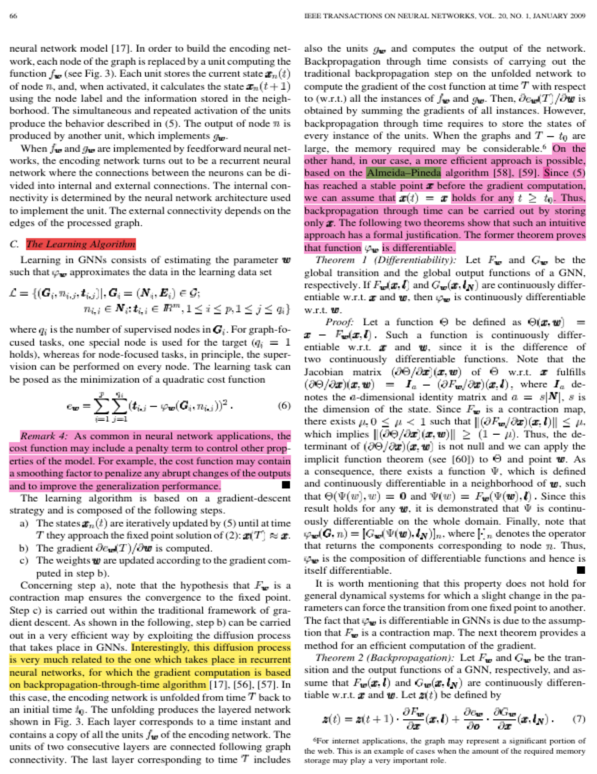
Each node's state evolves from x(t) to x(t+1) as the network processes information. The Global Update mechanism ensures this evolution happens synchronously across all nodes, maintaining consistency throughout the entire network. This process continues until the network reaches what the researchers call a "stable state" a point where further updates don't significantly change the nodes' states.



This advanced orchestration enables GNNs to capture both the local details of individual nodes and the broader patterns of the entire graph structure as they evolve over time. It's like watching a photograph develop, with each iteration, the picture becomes clearer and more detailed, until we have a complete understanding of the relationships within the data.

Diving Deep into GNN's Learning Process:

On page 66, the researchers unveiled the intricate details of how GNNs actually learn from data. This isn't just another neural network training process; it's a sophisticated two-phase approach that ensures the network can effectively process graph-structured information while maintaining stability.



66 IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 20, NO. 1, JANUARY 2009

neural network model [17]. In order to build the encoding network, each node of the graph is replaced by a unit computing the function $f_{\boldsymbol{w}}$ (see Fig. 3). Each unit stores the current state $\boldsymbol{x}_n(t)$ of node n, and, when activated, it calculates the state $\boldsymbol{x}_n(t+1)$ using the node label and the information stored in the neighborhood. The simultaneous and repeated activation of the units produce the behavior described in (5). The output of node n is produced by another unit, which implements $g_{\boldsymbol{w}}$.

When $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$ are implemented by feedforward neural networks, the encoding network turns out to be a recurrent neural network where the connections between the neurons can be divided into internal and external connections. The internal connectivity is determined by the neural network architecture used to implement the unit. The external connectivity depends on the edges of the processed graph.

C. The Learning Algorithm

Learning in GNNs consists of estimating the parameter $\boldsymbol{w}$ such that $\varphi_{\boldsymbol{w}}$ approximates the data in the learning data set

$$\mathcal{L} = \{(\boldsymbol{G}_i, n_{i,j}, \boldsymbol{t}_{i,j})|, \boldsymbol{G}_i = (\boldsymbol{N}_i, \boldsymbol{E}_i) \in \mathcal{G}; \; n_{i,j} \in \boldsymbol{N}_i; \boldsymbol{t}_{i,j} \in \mathbb{R}^m, 1 \le i \le p, 1 \le j \le q_i\}$$

where q_i is the number of supervised nodes in $\boldsymbol{G}_i$. For graph-focused tasks, one special node is used for the target ($q_i = 1$ holds), whereas for node-focused tasks, in principle, the supervision can be performed on every node. The learning task can be posed as the minimization of a quadratic cost function

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p}\sum_{j=1}^{q_i}(\boldsymbol{t}_{i,j} - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}))^2 . \qquad (6)$$

Remark 4: As common in neural network applications, the cost function may include a penalty term to control other properties of the model. For example, the cost function may contain a smoothing factor to penalize any abrupt changes of the outputs and to improve the generalization performance. ■

The learning algorithm is based on a gradient-descent strategy and is composed of the following steps.

a) The states $\boldsymbol{x}_n(t)$ are iteratively updated by (5) until at time T they approach the fixed point solution of (2): $\boldsymbol{x}(T) \approx \boldsymbol{x}$.
b) The gradient $\partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{w}$ is computed.
c) The weights $\boldsymbol{w}$ are updated according to the gradient computed in step b).

Concerning step a), note that the hypothesis that $F_{\boldsymbol{w}}$ is a contraction map ensures the convergence to the fixed point. Step c) is carried out within the traditional framework of gradient descent. As shown in the following, step b) can be carried out in a very efficient way by exploiting the diffusion process that takes place in GNNs. Interestingly, this diffusion process is very much related to the one which takes place in recurrent neural networks, for which the gradient computation is based on backpropagation-through-time algorithm [17], [56], [57]. In this case, the encoding network is unfolded from time T back to an initial time t_0. The unfolding produces the layered network shown in Fig. 3. Each layer corresponds to a time instant and contains a copy of all the units $f_{\boldsymbol{w}}$ of the encoding network. The units of two consecutive layers are connected following graph connectivity. The last layer corresponding to time T includes also the units $g_{\boldsymbol{w}}$ and computes the output of the network.

Backpropagation through time consists of carrying out the traditional backpropagation step on the unfolded network to compute the gradient of the cost function at time T with respect to (w.r.t.) all the instances of $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$. Then, $\partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{w}$ is obtained by summing the gradients of all instances. However, backpropagation through time requires to store the states of every instance of the units. When the graphs and $T - t_0$ are large, the memory required may be considerable.[6] On the other hand, in our case, a more efficient approach is possible, based on the Almeida–Pineda algorithm [58], [59]. Since (5) has reached a stable point $\boldsymbol{x}$ before the gradient computation, we can assume that $\boldsymbol{x}(t) = \boldsymbol{x}$ holds for any $t \ge t_0$. Thus, backpropagation through time can be carried out by storing only $\boldsymbol{x}$. The following two theorems show that such an intuitive approach has a formal justification. The former theorem proves that function $\varphi_{\boldsymbol{w}}$ is differentiable.

Theorem 1 (Differentiability): Let $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ be the global transition and the global output functions of a GNN, respectively. If $F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$ and $G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$ are continuously differentiable w.r.t. $\boldsymbol{x}$ and $\boldsymbol{w}$, then $\varphi_{\boldsymbol{w}}$ is continuously differentiable w.r.t. $\boldsymbol{w}$.

Proof: Let a function Θ be defined as $\Theta(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{x} - F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$. Such a function is continuously differentiable w.r.t. $\boldsymbol{x}$ and $\boldsymbol{w}$, since it is the difference of two continuously differentiable functions. Note that the Jacobian matrix $(\partial\Theta/\partial\boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})$ of Θ w.r.t. $\boldsymbol{x}$ fulfills $(\partial\Theta/\partial\boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{I}_a - (\partial F_{\boldsymbol{w}}/\partial\boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})$, where $\boldsymbol{I}_a$ denotes the a-dimensional identity matrix and $a = s|\boldsymbol{N}|$, s is the dimension of the state. Since $F_{\boldsymbol{w}}$ is a contraction map, there exists $\mu, 0 \le \mu < 1$ such that $\|(\partial F_{\boldsymbol{w}}/\partial\boldsymbol{x})(\boldsymbol{x}, \boldsymbol{l})\| \le \mu$, which implies $\|(\partial\Theta/\partial\boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})\| \ge (1 - \mu)$. Thus, the determinant of $(\partial\Theta/\partial\boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})$ is not null and we can apply the implicit function theorem (see [60]) to Θ and point $\boldsymbol{w}$. As a consequence, there exists a function Ψ, which is defined and continuously differentiable in a neighborhood of $\boldsymbol{w}$, such that $\Theta(\Psi(w), w) = \boldsymbol{0}$ and $\Psi(w) = F_{\boldsymbol{w}}(\Psi(\boldsymbol{w}), \boldsymbol{l})$. Since this result holds for any $\boldsymbol{w}$, it is demonstrated that Ψ is continuously differentiable on the whole domain. Finally, note that $\varphi_{\boldsymbol{w}}(\boldsymbol{G}, n) = [G_{\boldsymbol{w}}(\Psi(\boldsymbol{w}), \boldsymbol{l}_{\boldsymbol{N}})]_n$, where $[\cdot]_n$ denotes the operator that returns the components corresponding to node n. Thus, $\varphi_{\boldsymbol{w}}$ is the composition of differentiable functions and hence is itself differentiable. ■

It is worth mentioning that this property does not hold for general dynamical systems for which a slight change in the parameters can force the transition from one fixed point to another. The fact that $\varphi_{\boldsymbol{w}}$ is differentiable in GNNs is due to the assumption that $F_{\boldsymbol{w}}$ is a contraction map. The next theorem provides a method for an efficient computation of the gradient.

Theorem 2 (Backpropagation): Let $F_{\boldsymbol{w}}$ and $G_{\boldsymbol{w}}$ be the transition and the output functions of a GNN, respectively, and assume that $F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$ and $G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$ are continuously differentiable w.r.t. $\boldsymbol{x}$ and $\boldsymbol{w}$. Let $\boldsymbol{z}(t)$ be defined by

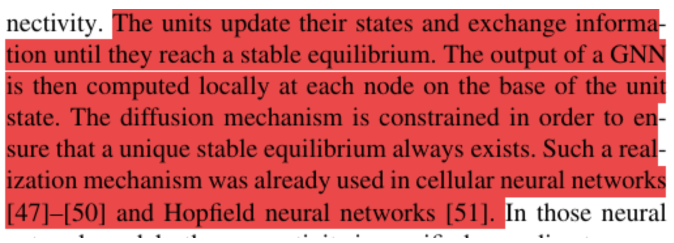
$$\boldsymbol{z}(t) = \boldsymbol{z}(t+1) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) . \qquad (7)$$

[6]For internet applications, the graph may represent a significant portion of the web. This is an example of cases when the amount of the required memory storage may play a very important role.

The Forward Phase

x(t) until ||x(t) - x(t-1)|| < εf

The network starts processing information at each node, then iteratively updates states until it reaches what the researchers call a "stable equilibrium." This isn't just a fancy term, it's a mathematically guaranteed state thanks to something called the Banach fixed point theorem.



nectivity. The units update their states and exchange information until they reach a stable equilibrium. The output of a GNN is then computed locally at each node on the base of the unit state. The diffusion mechanism is constrained in order to ensure that a unique stable equilibrium always exists. Such a realization mechanism was already used in cellular neural networks [47]–[50] and Hopfield neural networks [51]. In those neural

This is from p.63

During this phase, the network repeatedly updates node states x(t) until the difference between consecutive updates becomes negligibly small, that is, until ||x(t) - x(t-1)|| < εf. This mathematical condition ensures that the network has thoroughly processed all available information before moving to the next phase.

The Backward Phase

||z(t-1) - z(t)|| < εb

Here's where things get really interesting. The researchers implemented what they call the z-values computation, which moves backward through time. This process continues until the network reaches another stability condition: ||z(t-1) - z(t)|| < εb. They based this on the **Almeida-Pineda algorithm**, which ensures the network learns efficiently, regardless of its starting point.

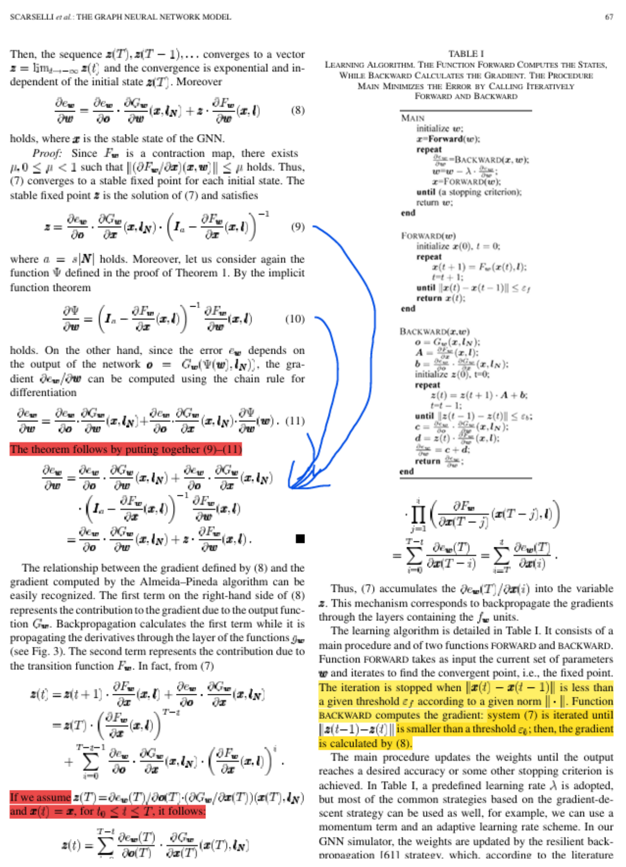
What makes this approach particularly elegant is its exponential convergence rate. This means that rather than slowly inching toward a solution, the network rapidly homes in on optimal parameters, making it both mathematically sophisticated and practically efficient. This two-phase design represents a crucial innovation in graph processing. By carefully balancing forward information flow with backward learning, the researchers created a system that could not only understand complex graph structures but also learn from them effectively.

* **εf and εb** are constant values used as learning rate, is different from problem to another problem. In general are be decimal values.

A Detailed Look at GNN's Training Flow

After exploring the basic mechanisms, learning algorithms, and all of the above, the researchers begin to outline how training works, starting on pages 66 and 67 of their paper, by providing a comprehensive flowchart that translates theoretical concepts into practical applications. This flowchart is not just a simple diagram; it is a detailed roadmap that shows exactly how large neural networks process and learn from graphically structured data.

The researchers break down the training process into seven carefully orchestrated stages, starting from the initialization of weights through to the final trained model. Before we dive into each stage of this process, it's important to understand that this flowchart represents a complete training cycle, something the researchers call an "epoch." Each cycle moves through forward computation, cost evaluation, gradient calculation, and weight updates, gradually refining the network's understanding of the graph structure.



SCARSELLI *et al.*: THE GRAPH NEURAL NETWORK MODEL 67

Then, the sequence $\boldsymbol{z}(T), \boldsymbol{z}(T-1), \ldots$ converges to a vector $\boldsymbol{z} = \lim_{t \to -\infty} \boldsymbol{z}(t)$ and the convergence is exponential and independent of the initial state $\boldsymbol{z}(T)$. Moreover

$$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) + \boldsymbol{z} \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \qquad (8)$$

holds, where $\boldsymbol{x}$ is the stable state of the GNN.

Proof: Since $F_{\boldsymbol{w}}$ is a contraction map, there exists $\mu, 0 \le \mu < 1$ such that $\|(\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x})(\boldsymbol{x}, \boldsymbol{w})\| \le \mu$ holds. Thus, (7) converges to a stable fixed point for each initial state. The stable fixed point $\boldsymbol{z}$ is the solution of (7) and satisfies

$$\boldsymbol{z} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) \cdot \left(\boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{-1} \qquad (9)$$

where $a = s|\boldsymbol{N}|$ holds. Moreover, let us consider again the function Ψ defined in the proof of Theorem 1. By the implicit function theorem

$$\frac{\partial \Psi}{\partial \boldsymbol{w}} = \left(\boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{-1} \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \qquad (10)$$

holds. On the other hand, since the error $e_{\boldsymbol{w}}$ depends on the output of the network $\boldsymbol{o} = G_{\boldsymbol{w}}(\Psi(\boldsymbol{w}), \boldsymbol{l}_{\boldsymbol{N}})$, the gradient $\partial e_{\boldsymbol{w}}/\partial \boldsymbol{w}$ can be computed using the chain rule for differentiation

$$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) \cdot \frac{\partial \Psi}{\partial \boldsymbol{w}}(\boldsymbol{w}). \qquad (11)$$

The theorem follows by putting together (9)–(11)

$$\begin{aligned}\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} &= \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) \\ &\quad \cdot \left(\boldsymbol{I}_a - \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{-1} \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) \\ &= \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) + \boldsymbol{z} \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}). \qquad \blacksquare\end{aligned}$$

The relationship between the gradient defined by (8) and the gradient computed by the Almeida–Pineda algorithm can be easily recognized. The first term on the right-hand side of (8) represents the contribution to the gradient due to the output function $G_{\boldsymbol{w}}$. Backpropagation calculates the first term while it is propagating the derivatives through the layer of the functions $g_{\boldsymbol{w}}$ (see Fig. 3). The second term represents the contribution due to the transition function $F_{\boldsymbol{w}}$. In fact, from (7)

$$\begin{aligned}\boldsymbol{z}(t) &= \boldsymbol{z}(t+1) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) \\ &= \boldsymbol{z}(T) \cdot \left(\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{T-t} \\ &\quad + \sum_{i=0}^{T-t-1} \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}}) \cdot \left(\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})\right)^{i}.\end{aligned}$$

If we assume $\boldsymbol{z}(T) = \partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{o}(T) \cdot (\partial G_{\boldsymbol{w}}/\partial \boldsymbol{x}(T))(\boldsymbol{x}(T), \boldsymbol{l}_{\boldsymbol{N}})$ and $\boldsymbol{x}(t) = \boldsymbol{x}$, for $t_0 \le t \le T$, it follows:

$$\begin{aligned}\boldsymbol{z}(t) &= \sum_{i=0}^{T-t} \frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{o}(T)} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}(T)}(\boldsymbol{x}(T), \boldsymbol{l}_{\boldsymbol{N}}) \\ &\quad \cdot \prod_{j=1}^{i} \left(\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}(T-j)}(\boldsymbol{x}(T-j), \boldsymbol{l})\right) \\ &= \sum_{i=0}^{T-t} \frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{x}(T-i)} = \sum_{i=T}^{t} \frac{\partial e_{\boldsymbol{w}}(T)}{\partial \boldsymbol{x}(i)}.\end{aligned}$$

TABLE I
LEARNING ALGORITHM. THE FUNCTION FORWARD COMPUTES THE STATES, WHILE BACKWARD CALCULATES THE GRADIENT. THE PROCEDURE MAIN MINIMIZES THE ERROR BY CALLING ITERATIVELY FORWARD AND BACKWARD

```
MAIN
    initialize $\boldsymbol{w}$;
    $\boldsymbol{x}$=Forward($\boldsymbol{w}$);
    repeat
        $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$=BACKWARD($\boldsymbol{x}, \boldsymbol{w}$);
        $\boldsymbol{w}=\boldsymbol{w} - \lambda \cdot \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$;
        $\boldsymbol{x}$=FORWARD($\boldsymbol{w}$);
    until (a stopping criterion);
    return $\boldsymbol{w}$;
end

FORWARD($\boldsymbol{w}$)
    initialize $\boldsymbol{x}(0)$, $t = 0$;
    repeat
        $\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l})$;
        $t=t+1$;
    until $\|\boldsymbol{x}(t) - \boldsymbol{x}(t-1)\| \le \varepsilon_f$
    return $\boldsymbol{x}(t)$;
end

BACKWARD($\boldsymbol{x}, \boldsymbol{w}$)
    $\boldsymbol{o} = G_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$;
    $\boldsymbol{A} = \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l})$;
    $\boldsymbol{b} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$;
    initialize $\boldsymbol{z}(0)$, t=0;
    repeat
        $\boldsymbol{z}(t) = \boldsymbol{z}(t+1) \cdot \boldsymbol{A} + \boldsymbol{b}$;
        $t=t-1$;
    until $\|\boldsymbol{z}(t-1) - \boldsymbol{z}(t)\| \le \varepsilon_b$;
    $\boldsymbol{c} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}_{\boldsymbol{N}})$;
    $\boldsymbol{d} = \boldsymbol{z}(t) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$;
    $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \boldsymbol{c} + \boldsymbol{d}$;
    return $\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}}$;
end
```

Thus, (7) accumulates the $\partial e_{\boldsymbol{w}}(T)/\partial \boldsymbol{x}(i)$ into the variable $\boldsymbol{z}$. This mechanism corresponds to backpropagate the gradients through the layers containing the $f_{\boldsymbol{w}}$ units.

The learning algorithm is detailed in Table I. It consists of a main procedure and of two functions FORWARD and BACKWARD. Function FORWARD takes as input the current set of parameters $\boldsymbol{w}$ and iterates to find the convergent point, i.e., the fixed point. The iteration is stopped when $\|\boldsymbol{x}(t) - \boldsymbol{x}(t-1)\|$ is less than a given threshold ε_f according to a given norm $\|\cdot\|$. Function BACKWARD computes the gradient: system (7) is iterated until $\|\boldsymbol{z}(t-1) - \boldsymbol{z}(t)\|$ is smaller than a threshold ε_b; then, the gradient is calculated by (8).

The main procedure updates the weights until the output reaches a desired accuracy or some other stopping criterion is achieved. In Table I, a predefined learning rate λ is adopted, but most of the common strategies based on the gradient-descent strategy can be used as well, for example, we can use a momentum term and an adaptive learning rate scheme. In our GNN simulator, the weights are updated by the resilient backpropagation [61] strategy, which, according to the literature

Below is a flowchart that integrates these equations into the GNN training process:

1. Start

- Initialize the graph with random weights *w*.

2. Forward Pass

- **Compute the state $\mathbf{X_n}$** for each node using the transition function(f_w)

$$\boldsymbol{x}_n(t+1) = f_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{\text{co}[n]}, \boldsymbol{x}_{\text{ne}[n]}(t), \boldsymbol{l}_{\text{ne}[n]})$$

***For general Compute the state $X_n(t)$**

$$\boldsymbol{x}(t+1) = F_{\boldsymbol{w}}(\boldsymbol{x}(t), \boldsymbol{l})$$

- **Compute the output $\mathbf{O_n}$** for each node using the output function:

$$\boldsymbol{o}_n(t) = g_{\boldsymbol{w}}(\boldsymbol{x}_n(t), \boldsymbol{l}_n), \qquad n \in \boldsymbol{N}.$$

- **Compute Cost Function (**Calculate the error between the predicted output and the target)

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p} \sum_{j=1}^{q_i} (\boldsymbol{t}_{i,j} - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}))^2. \qquad (6)$$

3.Backward Pass

- Compute the stable state Z using Equation

$$z = \frac{\partial e_w}{\partial o} \cdot \frac{\partial G_w}{\partial x}(x, l_N) \cdot \left(I_a - \frac{\partial F_w}{\partial x}(x, l) \right)^{-1}$$

- Compute the gradient of the cost function using:

$$\frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{w}} = \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l_N}) + \boldsymbol{z} \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l})$$

  - This equation computes the **total gradient** of the cost function with respect to the weights *ww*. It combines two terms:
    - The direct effect of the weights on the output (G_w).
    - The indirect effect of the weights on the state (F_w), propagated through the state variable Z.

- Update the state using the gradient:

$$\boldsymbol{z}(t) = \boldsymbol{z}(t+1) \cdot \frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l}) + \frac{\partial e_{\boldsymbol{w}}}{\partial \boldsymbol{o}} \cdot \frac{\partial G_{\boldsymbol{w}}}{\partial \boldsymbol{x}}(\boldsymbol{x}, \boldsymbol{l_N}) .$$

- This equation combines:
  - Local output effects: ∂cw/∂o · ∂Gw/∂x(x,IN)
  - Graph structure propagation: z(t+1) · ∂Fw/∂x(x,t)

4.Convergence Check

- Check if the state has converged to a stable point:

$$z = \lim_{t \to \infty} z(t)$$

by this roll :

$$\|\boldsymbol{z}(t-1) - \boldsymbol{z}(t)\| \leq \varepsilon_b;$$

*If not converged, repeat the forward and backward passes.

5.Update Weights

- Adjust the weights using the computed gradient:

W_{new}= W_{old} - λ · ∂ew/∂w

*Where λ is the learning rate.

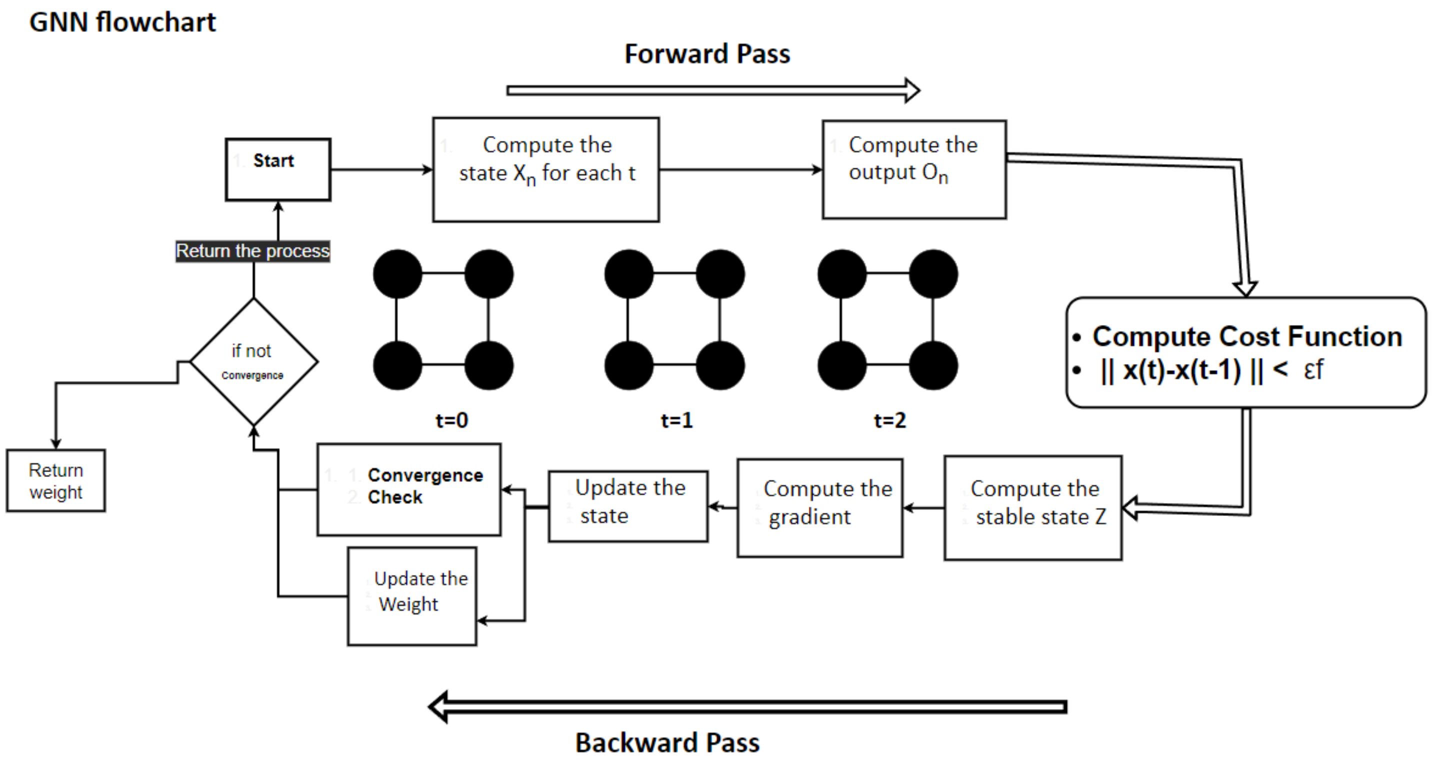
6.Repeat

- Repeat the forward pass, cost computation, backward pass, and weight update until convergence.

7.End

Explanation of the Flowchart

- **Forward Pass**: The network computes the state and output for each node based on the current weights.
- **Cost Function**: The error between the predicted output and the actual target is calculated.
- **Backward Pass**: The gradient of the cost function with respect to the weights is computed using Equations 8-12. This ensures that the network can learn effectively by adjusting its weights.
- **Convergence Check**: The network checks if the state has reached a stable point. If not, it repeats the process.
- **Weight Update**: The weights are adjusted to minimize the error, using gradient descent.



Two Approaches to Transition Functions

On page 68, the researchers mentioned how to implement the transition function that we explained earlier. It specifies how each node updates its state by collecting and processing information from its neighboring nodes. But here the researchers have devised ways to implement it, some of which were mentioned previously, so we will mention them in detail as stated on pages 64 and 68.

The researchers have identified **two main ways** to implement this function:

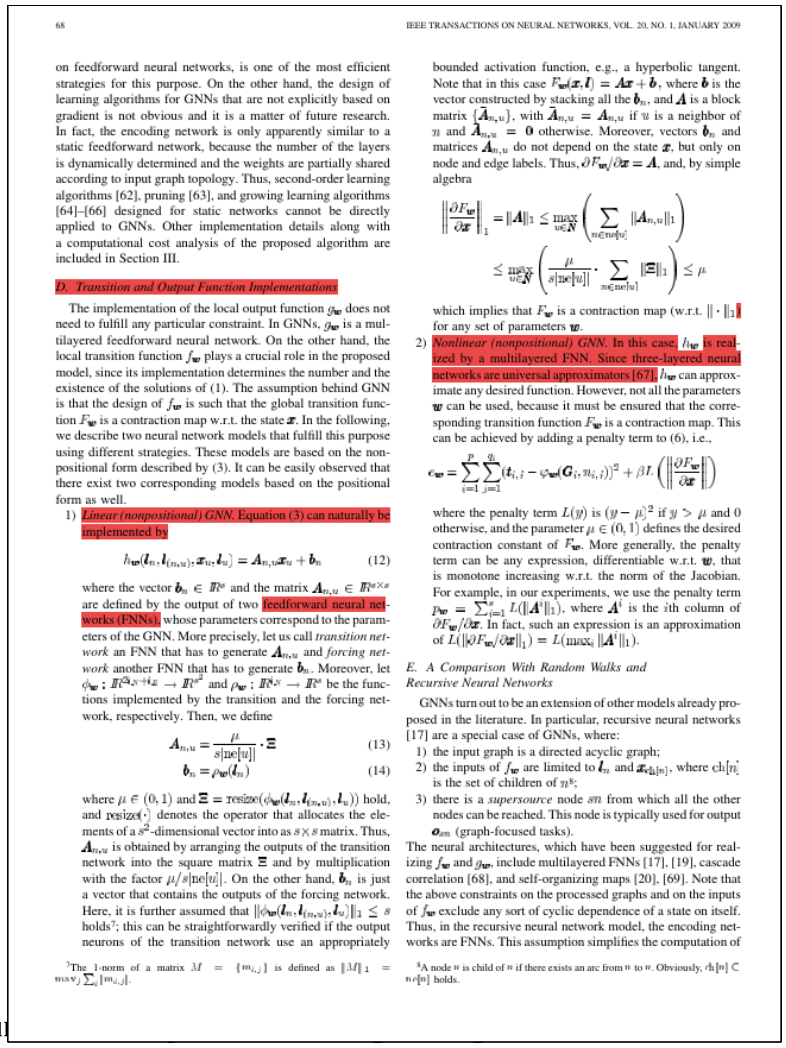
1. Local implementation (Equation 1 on page 64):

xn = fw(ln, lco[n], xne[n], lne[n])

Where:

- xn: State of node n
- ln: Label of node n
- lco[n]: Labels of edges connected to n
- xne[n]: States of neighboring nodes
- lne[n]: Labels of neighboring nodes

In this approach, the transition function takes into accoun
giving each neighbor a specific seat at the table - its location matters.



68 IEEE TRANSACTIONS ON NEURAL NETWORKS, VOL. 20, NO. 1, JANUARY 2009

on feedforward neural networks, is one of the most efficient strategies for this purpose. On the other hand, the design of learning algorithms for GNNs that are not explicitly based on gradient is not obvious and it is a matter of future research. In fact, the encoding network is only apparently similar to a static feedforward network, because the number of the layers is dynamically determined and the weights are partially shared according to input graph topology. Thus, second-order learning algorithms [62], pruning [63], and growing learning algorithms [64]–[66] designed for static networks cannot be directly applied to GNNs. Other implementation details along with a computational cost analysis of the proposed algorithm are included in Section III.

D. Transition and Output Function Implementations

The implementation of the local output function $g_{\boldsymbol{w}}$ does not need to fulfill any particular constraint. In GNNs, $g_{\boldsymbol{w}}$ is a multilayered feedforward neural network. On the other hand, the local transition function $f_{\boldsymbol{w}}$ plays a crucial role in the proposed model, since its implementation determines the number and the existence of the solutions of (1). The assumption behind GNN is that the design of $f_{\boldsymbol{w}}$ is such that the global transition function $F_{\boldsymbol{w}}$ is a contraction map w.r.t. the state $\boldsymbol{x}$. In the following, we describe two neural network models that fulfill this purpose using different strategies. These models are based on the nonpositional form described by (3). It can be easily observed that there exist two corresponding models based on the positional form as well.

1) *Linear (nonpositional) GNN.* Equation (3) can naturally be implemented by

$$h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) = \boldsymbol{A}_{n,u}\boldsymbol{x}_u + \boldsymbol{b}_n \qquad (12)$$

where the vector $\boldsymbol{b}_n \in \mathbb{R}^s$ and the matrix $\boldsymbol{A}_{n,u} \in \mathbb{R}^{s\times s}$ are defined by the output of two feedforward neural networks (FNNs), whose parameters correspond to the parameters of the GNN. More precisely, let us call *transition network* an FNN that has to generate $\boldsymbol{A}_{n,u}$ and *forcing network* another FNN that has to generate $\boldsymbol{b}_n$. Moreover, let $\phi_{\boldsymbol{w}} : \mathbb{R}^{2l_N+l_E} \rightarrow \mathbb{R}^{s^2}$ and $\rho_{\boldsymbol{w}} : \mathbb{R}^{l_N} \rightarrow \mathbb{R}^s$ be the functions implemented by the transition and the forcing network, respectively. Then, we define

$$\boldsymbol{A}_{n,u} = \frac{\mu}{s|\mathrm{ne}[u]|}\cdot\boldsymbol{\Xi} \qquad (13)$$

$$\boldsymbol{b}_n = \rho_{\boldsymbol{w}}(\boldsymbol{l}_n) \qquad (14)$$

where $\mu \in (0,1)$ and $\boldsymbol{\Xi} = \mathrm{resize}(\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u))$ hold, and resize$(\cdot)$ denotes the operator that allocates the elements of a s^2-dimensional vector into as $s \times s$ matrix. Thus, $\boldsymbol{A}_{n,u}$ is obtained by arranging the outputs of the transition network into the square matrix $\boldsymbol{\Xi}$ and by multiplication with the factor $\mu/s|\mathrm{ne}[u]|$. On the other hand, $\boldsymbol{b}_n$ is just a vector that contains the outputs of the forcing network. Here, it is further assumed that $\|\phi_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{l}_u)\|_1 \leq s$ holds[7]; this can be straightforwardly verified if the output neurons of the transition network use an appropriately bounded activation function, e.g., a hyperbolic tangent. Note that in this case $F_{\boldsymbol{w}}(\boldsymbol{x}, \boldsymbol{l}) = \boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{b}$ is the vector constructed by stacking all the $\boldsymbol{b}_n$, and $\boldsymbol{A}$ is a block matrix $\{\bar{\boldsymbol{A}}_{n,u}\}$, with $\bar{\boldsymbol{A}}_{n,u} = \boldsymbol{A}_{n,u}$ if u is a neighbor of n and $\bar{\boldsymbol{A}}_{n,u} = \boldsymbol{0}$ otherwise. Moreover, vectors $\boldsymbol{b}_n$ and matrices $\boldsymbol{A}_{n,u}$ do not depend on the state $\boldsymbol{x}$, but only on node and edge labels. Thus, $\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x} = \boldsymbol{A}$, and, by simple algebra

$$\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|_1 = \|\boldsymbol{A}\|_1 \leq \max_{u\in\boldsymbol{N}}\left(\sum_{n\in\mathrm{ne}[u]}\|\boldsymbol{A}_{n,u}\|_1\right) \leq \max_{u\in\boldsymbol{N}}\left(\frac{\mu}{s|\mathrm{ne}[u]|}\cdot\sum_{n\in\mathrm{ne}[u]}\|\boldsymbol{\Xi}\|_1\right) \leq \mu$$

which implies that $F_{\boldsymbol{w}}$ is a contraction map (w.r.t. $\|\cdot\|_1$) for any set of parameters $\boldsymbol{w}$.

2) *Nonlinear (nonpositional) GNN.* In this case, $h_{\boldsymbol{w}}$ is realized by a multilayered FNN. Since three-layered neural networks are universal approximators [67], $h_{\boldsymbol{w}}$ can approximate any desired function. However, not all the parameters $\boldsymbol{w}$ can be used, because it must be ensured that the corresponding transition function $F_{\boldsymbol{w}}$ is a contraction map. This can be achieved by adding a penalty term to (6), i.e.,

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p}\sum_{j=1}^{q_i}(\boldsymbol{t}_{i,j} - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}))^2 + \beta L\left(\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|\right)$$

where the penalty term $L(y)$ is $(y-\mu)^2$ if $y > \mu$ and 0 otherwise, and the parameter $\mu \in (0,1)$ defines the desired contraction constant of $F_{\boldsymbol{w}}$. More generally, the penalty term can be any expression, differentiable w.r.t. $\boldsymbol{w}$, that is monotone increasing w.r.t. the norm of the Jacobian. For example, in our experiments, we use the penalty term $p_{\boldsymbol{w}} = \sum_{i=1}^{s} L(\|\boldsymbol{A}^i\|_1)$, where $\boldsymbol{A}^i$ is the ith column of $\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x}$. In fact, such an expression is an approximation of $L(\|\partial F_{\boldsymbol{w}}/\partial \boldsymbol{x}\|_1) = L(\max_i \|\boldsymbol{A}^i\|_1)$.

E. A Comparison With Random Walks and Recursive Neural Networks

GNNs turn out to be an extension of other models already proposed in the literature. In particular, recursive neural networks [17] are a special case of GNNs, where:

1) the input graph is a directed acyclic graph;
2) the inputs of $f_{\boldsymbol{w}}$ are limited to $\boldsymbol{l}_n$ and $\boldsymbol{x}_{\mathrm{ch}[n]}$, where $\mathrm{ch}[n]$ is the set of children of n[8];
3) there is a *supersource* node sn from which all the other nodes can be reached. This node is typically used for output $\boldsymbol{o}_{sn}$ (graph-focused tasks).

The neural architectures, which have been suggested for realizing $f_{\boldsymbol{w}}$ and $g_{\boldsymbol{w}}$, include multilayered FNNs [17], [19], cascade correlation [68], and self-organizing maps [20], [69]. Note that the above constraints on the processed graphs and on the inputs of $f_{\boldsymbol{w}}$ exclude any sort of cyclic dependence of a state on itself. Thus, in the recursive neural network model, the encoding networks are FNNs. This assumption simplifies the computation of

[7]The 1-norm of a matrix $M = \{m_{i,j}\}$ is defined as $\|M\|_1 = \max_j \sum_i |m_{i,j}|$.

[8]A node u is child of n if there exists an arc from n to u. Obviously, $\mathrm{ch}[n] \subseteq \mathrm{ne}[n]$ holds.

2. Non-local implementation (Eq. 3, p. 64 and p. 68):

This version is more flexible and does not care about the specific positions of neighbors. It treats all neighbors equally, like guests at a round table where the seating order does not matter. The function collects information from all neighbors uniformly.

$$\boldsymbol{x}_n = \sum_{u\in\mathrm{ne}[n]} h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u), \qquad n \in \boldsymbol{N} \qquad (3)$$

The researchers describe two ways to implement the nonlocal version (Equation 3):

- Linear implementation:

$$h_{\boldsymbol{w}}(\boldsymbol{l}_n, \boldsymbol{l}_{(n,u)}, \boldsymbol{x}_u, \boldsymbol{l}_u) = \boldsymbol{A}_{n,u}\boldsymbol{x}_u + \boldsymbol{b}_n \qquad (12)$$

  An = σφ(φw(ln, lN(n), lE(n)))

  bn = ρw(ln, lN(n), lE(n))

  With stability constraint: ||An||1 ≤ β, 0 < β < 1

- Uses two separate neural networks: a transition network and a forcing network

- The transition network determines how the states of neighbors affect the node

- The forcing network modifies the state of the node with a bias term

- This approach ensures mathematical stability through controlled information flow

- Nonlinear implementation:

$$e_{\boldsymbol{w}} = \sum_{i=1}^{p}\sum_{j=1}^{q_i}(\boldsymbol{t}_{i,j} - \varphi_{\boldsymbol{w}}(\boldsymbol{G}_i, n_{i,j}))^2 + \beta L\left(\left\|\frac{\partial F_{\boldsymbol{w}}}{\partial \boldsymbol{x}}\right\|\right)$$

  - Uses a single multilayer neural network

  - More flexible in learning complex relationships

  - Requires careful training to maintain stability

  - Uses a penalty term during training to prevent unstable behavior

Both implementations serve the same purpose but offer different trade-offs between computational efficiency and expressive power.

The Original GNN: Laying the Foundation for Modern GNNs

The 2009 GNN model by Scarselli and colleagues may not be running in today's systems, but it sparked development. Think of it as a basic template that all future versions would build on without sacrificing the core idea. So, this post introduced fundamental ideas like how nodes share information and learn from their neighbors, concepts that every modern GNN still uses today.

One of its most successful descendants is **GraphSAGE**, which powers **Pinterest's recommendation** engine. When you click "More of this" on Pinterest, you see GraphSAGE in action, processing a massive network of 3 billion pins and 18 billion connections to find what might interest you next. This evolution from theoretical framework to practical applications shows how far we've come. While the original model laid the foundation, modern variants of it now tackle real-world challenges on unprecedented scales.

In conclusion, the diving deep into the practical side of Graph Neural Networks, we've seen how elegant mathematical principles become powerful real-world tools. The researchers built something remarkable here, they found a way to make neural networks truly understand and work with graph structures. Whether using the straightforward linear approach or the more flexible nonlinear method, GNNs show us how complex data relationships can be learned and processed effectively. What makes this work particularly exciting is how it bridges theory and practice. The careful balance between information flow and learning stability opens doors for using GNNs in everything from analyzing molecules to understanding social networks. Looking ahead, this practical foundation gives us not just working solutions for today's challenges, but a framework that will keep evolving as we push the boundaries of what artificial intelligence can do with interconnected data.

Source:
The Graph Neural Network Model | IEEE Journals & Magazine | IEEE Xplore

Graph Neural Networks Theoretical Foundations and Core Mechanisms-Part1.pdf