Graph Neural Networks for Small Graph and Giant Network Representation Learning

Jiawei Zhang JIAWEI@IFMLAB.ORG
Founder and Director
Information Fusion and Mining Laboratory
(First Version: July 2019; Revision: July 2019.)

Abstract

Graph neural networks denote a group of neural network models introduced for the representation learning tasks on graph data specifically. Graph neural networks have been demonstrated to be effective for capturing network structure information, and the learned representations can achieve the state-of-the-art performance on node and graph classification tasks. Besides the different application scenarios, the architectures of graph neural network models also depend on the studied graph types a lot. Graph data studied in research can be generally categorized into two main types, i.e., small graphs *vs.* giant networks, which differ from each other a lot in the *size*, *instance number* and *label annotation*. Several different types of graph neural network models have been introduced for learning the representations from such different types of graphs already. In this paper, for these two different types of graph data, we will introduce the graph neural networks introduced in recent years. To be more specific, the graph neural networks introduced in this paper include IsoNN [4], SDBN [7], LF&ER [6], GCN [3], GAT [5], DifNN [8], GNL [1], GraphSage [2] and seGEN [9]. Among these graph neural network models, IsoNN, SDBN and LF&ER are initially proposed for small graphs and the remaining ones are initially proposed for giant networks instead. The readers are also suggested to refer to these papers for detailed information when reading this tutorial paper.

Keywords: Graph Neural Network; Representation Learning; Graph Mining; Deep Learning

Contents

1 Introduction **2**

2 Graph Neural Networks for Small Graphs **3**
2.1 IsoNN: Isomorphic Neural Network 4
2.1.1 Graph Isomorphic Layer 5
2.1.2 Min-pooling Layers 6
2.1.3 Classification Component 6
2.2 SDBN: Structural Deep Brain Network 7
2.2.1 Graph Reordering 7
2.2.2 Structural Augmentation 8
2.2.3 Learning of the SDBN Model 8
2.3 LF&ER: Deep Autoencoder based Latent Feature Extraction 10

2.3.1 Deep Autoencoder Model 11
2.3.2 Latent Representation Learning 11

3 Graph Neural Networks for Giant Networks 12
3.1 GCN: Graph Convolutional Network 12
3.1.1 Spectral Graph Convolution 13
3.1.2 Graph Convolution Approximation 13
3.1.3 Deep Graph Convolutional Network Learning 14
3.2 GAT: Graph Attention Network 15
3.2.1 Graph Attention Coefficient Computation 15
3.2.2 Representation Update via Neighborhood Aggregation 16
3.2.3 Multi-Head Attention Aggregation 16
3.3 DifNN: Deep Diffusive Neural Network 17
3.3.1 Model Architecture 17
3.3.2 Gated Diffusive Unit 18
3.3.3 DifNN Model Learning 19
3.4 GNL: Graph Neural Lasso 20
3.4.1 Dynamic Gated Diffusive Unit 20
3.4.2 Attentive Neighborhood Influence Aggregation 21
3.4.3 Graph Neural Lasso Model Learning 22
3.5 GraphSage: Graph Sample and Aggregate 23
3.5.1 Framework Descriptions 23
3.5.2 Aggregator Function 24
3.6 seGEN: Sample and Ensemble Genetic Evolutionary Network 24
3.6.1 seGEN Architecture Description 25
3.6.2 Genetic Evolutionary Learning of seGEN 26
3.6.3 Result Ensemble 27

4 Challenges and Opportunities 28
4.1 Over-Smoothing Problem 28
4.2 Optimization Efficiency 28
4.3 The Gap Between Small Graphs and Giant Networks 29
4.4 Graph Neural Network for Dynamic Networks 29
4.5 Graph Neural Network for Complex Networks 29

5 Summary 29

1. Introduction

In the era of big data, graph provides a generalized representation of many different types of inter-connected data collected from various disciplines. Besides the unique attributes possessed by individual nodes, the extensive connections among the nodes can convey very complex yet important information. Graph data are very difficult to deal with because of their *various shapes* (e.g., small brain graphs *vs.* giant online social networks), *complex structures* (containing various kinds of nodes and extensive connections) and *diverse attributes* (attached to the nodes and links). Great challenges exist in handling the graph data with traditional machine learning algorithms directly, which usually take feature vectors as the input. Viewed in such a perspective, learning the feature vector representations of graph data will be an important problem.

The graph data studied in research can be generally categorized into two main types, i.e., small graphs *vs.* giant networks, which differ from each other a lot in the *size*, *instance number* and *label annotation*.

- The small graphs we study are generally of a much smaller size, but with a large number of instances, and each graph instance is annotated with certain labels. The representative examples include the human brain graph, molecular graph, and real-estate community graph, whose nodes (usually only in hundreds) represent the brain regions, atoms and POIs, respectively.
- On the contrary, giant networks in research usually involve a large number of nodes/links, but with only one single network instance, and individual nodes are labeled instead of the network. Examples of giant networks include social network (e.g., Facebook), eCommerce network (e.g., Amazon) and bibliographic network (e.g., DBLP), which all contain millions even billions of nodes.

Due to these property distinctions, the representation learning algorithms proposed for small graphs and giant networks are very different. To solve the small graph oriented problems, the existing graph neural networks focus on learning a representation of the whole graph (not the individual nodes) based on the graph structure and attribute information. Several different graph neural network models have been introduced already, including IsoNN (Isomorphic Neural Network) [4], SDBN (Structural Deep Brain Network) [7] and LF&ER (Deep Autoencoder based Latent Feature Extraction) [6]. These models are proposed for different small graph oriented application scenarios, covering both brain graphs and community POI graphs, which can also be applied to other application settings with minor extensions.

Meanwhile, for the giant network studies, in recent years, many research works propose to apply graph neural networks to learn their low-dimensional feature representations, where each node is represented as a feature vector. With these learned node representations, the graph neural network model can directly infer the potential labels of the nodes/links. To achieve such objectives, several different type of graph neural network models have been introduced, including GCN (Graph Convolutional Network) [3], GAT (Graph Attention Network) [5], DifNN (Deep Diffusive Neural Network) [8], GNL (Graph Neural Lasso), GraphSage (Graph Sample and Aggregate) [2] and seGEN (Sample and Ensemble Genetic Evolutionary Network) [9].

In this paper, we will introduce the aforementioned graph neural networks proposed for small graphs and giant networks, respectively. This tutorial paper will be updated accordingly as we observe the latest developments on this topic.

2. Graph Neural Networks for Small Graphs

In this section, we will introduce the graph neural networks proposed for the representation learning tasks on small graphs. Formally, we can represent the set of small graphs to be

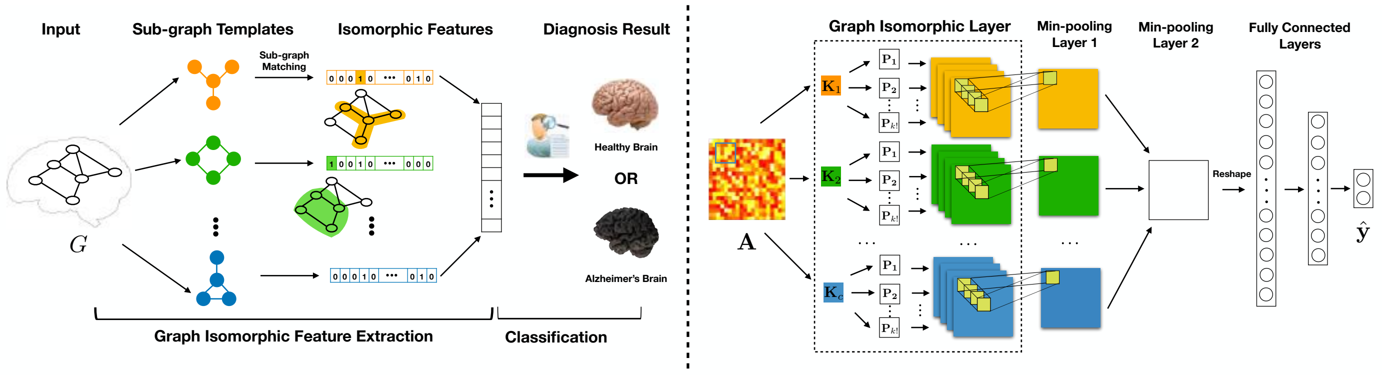


Figure 1: IsoNN Model Architecture [4]. (The left plot provides the outline of IsoNN, including the *graph isomorphic feature extraction* and *classification* components. The right plot illustrates the detailed architecture of the proposed model, where the *graph isomorphic features* are extracted with the *graph isomorphic layer* and two *min-pooling layers*, and the graphs are further classified with three *fully-connected layers*.)

studied in this section as $\mathcal{G} = (G_1, \mathbf{y}_1), (G_2, \mathbf{y}_2), \cdots, (G_n, \mathbf{y}_n)$, where $G_i = (\mathcal{V}_i, \mathcal{E}_i)$ denotes a small graph instance and $\mathbf{y}_i \in \mathbb{R}^{d_y}$ denotes its label vector. Given a graph $G_i \in \mathcal{G}$, we can denote its network size as the number of involved nodes, i.e., $|\mathcal{V}_i|$. Normally, the small graphs to be studied in set $\mathcal{G}$ are of the same size. Meanwhile, depending on the application scenarios, the objective labels of the graph instances can be binary-class/multi-class vectors. The small graph oriented graph neural networks aim at learning a mapping, i.e., $f : \mathcal{G} \to \mathbb{R}^{d_h}$, to project the graph instances to their feature vector representations, which will be further utilized to infer their corresponding labels. Specifically, the graph neural network models to be introduced in this section include IsoNN [4], SDBN [7] and LF&ER [6]. The readers are also suggested to refer to these papers for detailed information when reading this tutorial paper.

2.1 IsoNN: Isomorphic Neural Network

Graph isomorphic neural network (IsoNN) proposed in [4] recently aims at extracting meaningful sub-graph patterns from the input graph for representation learning. Sub-graph mining techniques have been demonstrated to be effective for feature extraction in the existing works. Instead of designing the sub-graph templates manually, IsoNN proposes to integrate the sub-graph based feature extraction approaches into the neural network framework for automatic feature representation learning. As illustrated in Figure 1, IsoNN includes two main components: *graph Isomorphic feature extraction* component and *classification* component. IsoNN can be in a deeper architecture by involving multiple graph isomorphic feature extraction components so that the model will learn more complex sub-graph patterns.

The graph isomorphic feature extraction component in IsoNN targets at the automatic sub-graph pattern learning and brain graph feature extraction with the following three layers:*graph isomorphic layer*, *min-pooling layer 1* and *min-pooling layer 2*, which will be introduced as follows, respectively. Meanwhile, the classification component used in IsoNN involves several fully connected layers, which project the learned isomorphic features to the corresponding graph labels.

2.1.1 Graph Isomorphic Layer

In IsoNN, the sub-graph based feature extraction process is achieved by a novel *graph isomorphic layer*. Formally, given a brain graph $G = (\mathcal{V}, \mathcal{E})$, its adjacency matrix can be represented as $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$. In order to find the existence of specific sub-graph patterns in the input graph, IsoNN matches the input graph with a set of sub-graph templates. Instead of defining these sub-graph templates manually as the existing works, each template is denoted as a kernel variable $\mathbf{K}_i \in \mathbb{R}^{k \times k}, \forall i \in \{1, 2, \cdots, c\}$ and IsoNN will learn these kernel variables automatically. Here, k denotes the node number in the templates and c is the channel number. Meanwhile, to match one template (i.e., the kernel variable matrix $\mathbf{K}_i$) with regions in the input graph (i.e., sub-matrices in $\mathbf{A}$), IsoNN uses a set of permutation matrices, which map both rows and columns of the kernel matrix to the sub-matrices in $\mathbf{A}$ effectively. The permutation matrix can be represented as $\mathbf{P} \in \{0,1\}^{k \times k}$ that shares the same dimensions with the kernel matrix. Given a kernel variable matrix $\mathbf{K}_i$ and a regional sub-matrix $\mathbf{M}_{(s,t)} \in \mathbb{R}^{k \times k}$ in $\mathbf{A}$ (where $\mathbf{M}_{(s,t)}(1:k, 1:k) = \mathbf{A}(s : s+k-1, t : t+k-1)$ and index pair $s, t \in \{1, 2, \cdots, (|\mathcal{V}| - k + 1)\}$), there may exist $k!$ different such permutation matrices and the optimal one can be denoted as $\mathbf{P}^*$:

$$\mathbf{P}^* = \arg\min_{\mathbf{P} \in \mathcal{P}} \left\| \mathbf{P}\mathbf{K}_i\mathbf{P}^\top - \mathbf{M}_{(s,t)} \right\|_F^2, \tag{1}$$

where $\mathcal{P} = \{\mathbf{P}_1, \mathbf{P}_2, \cdots, \mathbf{P}_{k!}\}$ covers all the potential permutation matrices. The F-norm term measures the mapping loss, which is also used as the graph isomorphic feature in IsoNN. Formally, the isomorphic feature extracted based on the kernel $\mathbf{K}_i$ for the regional sub-matrix $\mathbf{M}_{(s,t)}$ in $\mathbf{A}$ can be represented as

$$\begin{aligned} z_{i,(s,t)} &= \left\| \mathbf{P}^*\mathbf{K}_i(\mathbf{P}^*)^\top - \mathbf{M}_{(s,t)} \right\|_F^2 \\ &= \min \left\{ \left\| \mathbf{P}\mathbf{K}_i\mathbf{P}^\top - \mathbf{M}_{(s,t)} \right\|_F^2 \right\}_{\mathbf{P} \in \mathcal{P}} \\ &= \min(\bar{\mathbf{z}}_{i,(s,t)}(1:k!)), \end{aligned} \tag{2}$$

where vector $\bar{\mathbf{z}}_{i,(s,t)} \in \mathbb{R}^{k!}$ with $\bar{\mathbf{z}}_{i,(s,t)}(j) = \left\| \mathbf{P}_j\mathbf{K}_i\mathbf{P}_j^\top - \mathbf{M}_{(s,t)} \right\|_F^2, \forall j \in \{1, 2, \cdots, k!\}$ denoting the features computed on the permutation matrix $\mathbf{P}_j \in \mathcal{P}$. Furthermore, by shifting the kernel matrix $\mathbf{K}_i$ on regional sub-matrices in $\mathbf{A}$, the isomorphic features extracted by IsoNN from the input graph can be denoted as a 3-way tensor $\mathcal{Z}_i \in \mathbb{R}^{k! \times (|\mathcal{V}|-k+1) \times (|\mathcal{V}|-k+1)}$, where $\mathcal{Z}_i(1:k!, s, t) = \bar{\mathbf{z}}_{i,(s,t)}(1:k!)$.

2.1.2 Min-pooling Layers

• **Min-pooling Layer 1**: As indicated by the Figure 1, IsoNN computes the final isomorphic features with the optimal permutation matrix for the kernel $\mathbf{K}_i$ via two steps: (1) computing all the potential isomorphic features via different permutation matrices with the graph isomorphic layer, and (2) identifying the optimal features with the min-pooling layer 1 and layer 2. Formally, given the tensor $\mathcal{Z}_i$ computed by $\mathbf{K}_i$ in the graph isomorphic layer, IsoNN will identify the optimal permutation matrices via the min-pooling layer 1. From tensor $\mathcal{Z}_i$, the features computed with the optimal permutation matrices can be denoted as $\mathbf{Z}_i$, where

$$\mathbf{Z}_i(s,t) = \min\{\mathcal{Z}_i(1:k!, s, t)\}, \forall s,t \in \{1, 2, \cdots, (|\mathcal{V}| - k + 1)\}. \tag{3}$$

The min-pooling layer 1 learns the optimal feature matrix $\mathbf{Z}_i$ for kernel $\mathbf{K}_i$ along the first dimension of tensor $\mathcal{Z}_i$, which are computed by the optimal permutation matrices. In a similar way, for the remaining kernels, their optimal graph isomorphic features can be obtained and denoted as matrices $\mathbf{Z}_1$, $\mathbf{Z}_2$, $\cdots$, $\mathbf{Z}_c$, respectively.

• **Min-pooling Layer 2**: For the same region in the input graph, different kernels can be applied to match the regional sub-matrix. Inspired by this, IsoNN incorporates the min-pooling layer 2, so that the model can find the best kernels that match the regions in $\mathbf{A}$. With inputs $\mathbf{Z}_1, \mathbf{Z}_2, \cdots, \mathbf{Z}_c$, the min-pooling layer 2 in IsoNN can identify the optimal features across all the kernels, which can be denoted as matrix $\mathbf{Q}$ with

$$\mathbf{Q}(s,t) = min\{\mathbf{Z}_1(s,t), \mathbf{Z}_2(s,t), \cdots, \mathbf{Z}_c(s,t)\}, \forall s,t \in \{1, 2, \cdots, (|\mathcal{V}| - k + 1)\}. \tag{4}$$

Entry $\mathbf{Q}(s,t)$ denotes the graph isomorphic feature computed by the best sub-graph kernel on the regional matrix $\mathbf{M}_{(s,t)}$ in $\mathbf{A}$. Thus, via min-pooling layer 2, let $\mathbf{Q}$ be the final isomorphic feature matrix, which preserves the best sub-graph patterns contributing to the classification result. In addition, min-pooling layer 2 also effectively shrinks the feature length and greatly reduces the number of variables to be learned in the following classification component.

2.1.3 Classification Component

Given a brain graph instance $G_g \in \mathcal{B}$ ($\mathcal{B} \subset \mathcal{T}$ denotes the training batch), its extracted isomorphic feature matrix can be denoted as $\mathbf{Q}_g$. By feeding its flat vectorized representation vector $\mathbf{q}_g = vec(\mathbf{Q}_g)$ as the input into the classification component (with three fully-connected layers), the predicted label vector by IsoNN on the instance can be represented as $\hat{\mathbf{y}}_g$. Several frequently used loss functions, e.g., cross-entropy, can be used to measure the introduced loss between $\hat{\mathbf{y}}_g$ and the ground-truth label vector $\mathbf{y}_g$. Formally, the fully-connected (FC) layers and the loss function used in IsoNN can be represented as follows:

$$\text{FC Layers: } \begin{cases} \mathbf{d}_1 &= \sigma(\mathbf{W}_1\mathbf{q}_g + \mathbf{b}_1), \\ \mathbf{d}_2 &= \sigma(\mathbf{W}_2\mathbf{d}_1 + \mathbf{b}_2), \\ \hat{\mathbf{y}}_g &= softmax(\mathbf{W}_3\mathbf{d}_2 + \mathbf{b}_3); \end{cases} \tag{5}$$

and

$$\text{Loss Function: } \ell(\mathbf{\Theta}) = -\sum_{g \in \mathcal{B}} \sum_j \mathbf{y}_g(j) \log \hat{\mathbf{y}}_g(j), \tag{6}$$

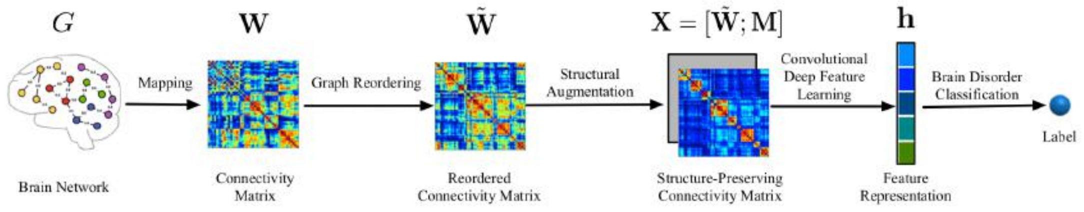

G
$\mathbf{W}$
$\tilde{\mathbf{W}}$
$\mathbf{X} = [\tilde{\mathbf{W}}; \mathbf{M}]$
$\mathbf{h}$
Mapping
Graph Reordering
Structural Augmentation
Convolutional Deep Feature Learning
Brain Disorder Classification
Label
Brain Network
Connectivity Matrix
Reordered Connectivity Matrix
Structure-Preserving Connectivity Matrix
Feature Representation

Figure 2: SDBN Model Architecture [7].

where $\mathbf{W}_i$ and $\mathbf{b}_i$ are the weight and biase in i_{th} layer, $\sigma(\cdot)$ denotes the sigmoid activation function and $softmax(\cdot)$ is the softmax function for output normalization. Variables $\mathbf{\Theta} = (\{\mathbf{K}\}_{i=1}^{k}, \{\mathbf{W}_i, \mathbf{b}_i\}_{i=1}^{3})$ (including the kernel matrices and weight/bias terms) involved in the model can be effectively learned with the error back propagation algorithm by minimizing the above loss function. For more information about IsoNN, the readers are suggested to refer to [4] for detailed descriptions.

2.2 SDBN: Structural Deep Brain Network

Structural Deep Brain Network (SDBN) initially proposed [7] applies the deep convolutional neural network to the brain graph mining problem, which can be viewed as an integration of convolutional neural network and autoencoder. Similar to IsoNN, SDBN also observes the order-less property with the brain graph data, and introduce a graph reordering approach to resolve the problem.

As illustrated in Figure 2, besides the necessary graph data processing and representation, SDBN involves three main steps to learn the final representations and labels fo the graph instances, i.e., *graph reordering*, *structural augmentation* and *convolutional feature learning*, which will be introduced as follows, respectively.

2.2.1 Graph Reordering

Given the graph set $\mathcal{G} = \{G_1, G_2, \cdots, G_n\}$, the goal of graph reordering is to find a node labeling ℓ_n such that for any two graphs $G_i, G_j \in \mathcal{G}$ randomly draw from $\mathcal{G}$, the expected differences between the distance of the graph connectivity adjacency matrices based on ℓ_n and the distance of the graphs in the graph space is minimized. Formally, for each graph instance $G_i \in \mathcal{G}$, its connectivity adjacency matrix can be denoted as $\mathbf{A}_i$. Let $d_{\mathcal{A}}$ and $d_{\mathcal{G}}$ denote the distance metrics on the adjacency matrix domain $\mathcal{A}$ and graph domain $\mathcal{G}$ respectively, the graph reordering problem can be formulated as the following optimization problem:

$$\arg\min_{\ell_n} \mathbb{E}_{G_i, G_j \in \mathcal{G}} \left(\| d_{\mathcal{A}}(\mathbf{A}_i, \mathbf{A}_j) - d_{\mathcal{G}}(G_i, G_j) \| \right). \tag{7}$$

Graph reordering is a combinatorial optimization problem, which has also be demonstrated to be NP-hard and is computationally infeasible to address in polynomial time. SDBN proposes to apply the spectral clustering to help reorder the nodes and brain graph connectivity instead. Formally, based on the brain graph adjacency matrix $\mathbf{A}_i$ of G_i, its corresponding Laplacian matrix can be represented as $\mathbf{L}_i$. The spectral clustering algorithm aims at partitioning the brain graph G_i into K modules, where the node-module belonging relationships are denoted by matrix $\mathbf{F} \in \mathbb{R}^{|\mathcal{V}| \times K}$. The optimal $\mathbf{F}$ can be effectively learned with the following objective function:

$$\min_{\mathbf{F}} tr\left(\mathbf{F}^\top \mathbf{L}_i \mathbf{F}\right) \quad s.t. \mathbf{F}^\top \mathbf{F} = \mathbf{I}, \tag{8}$$

where $\mathbf{I} \in \{0,1\}^{K \times K}$ denotes an identity matrix and the constraint is added to ensure one node is assigned to one module only. From the learned optimal $\mathbf{F}$, SDBN can assign the nodes in graph G_i to their modules $\mathcal{M} = \{M_1, M_2, \cdots, M_K\}$, where $\mathcal{V} = \bigcup_{i=1}^{K} M_i$ and $M_i \cap M_j = \emptyset, \forall i \neq j$ and $i, j \in \{1, 2, \cdots, K\}$. Such learned modules $\mathcal{M}$ can help reorder the nodes in the graph into relatively compact regions, and the graph connectivity adjacency matrix $\mathbf{A}_i$ after reordering can be denoted as $\tilde{\mathbf{A}}_i$. Similar operations will be performed on the other graph instances in the set $\mathcal{G}$.

2.2.2 Structural Augmentation

Prior to feeding the reordered graph adjacency matrix to the deep convolutional neural network for representation learning, SDBN proposes to augment the network structure by both refining the connectivity adjacency matrix and creating an additional module identification channel.

- **Reordered Adjacency Matrix Refinement**: Formally, for graph $G_i \in \mathcal{G}$, based on its reordered adjacency matrix $\tilde{\mathbf{A}}_i$ obtained from the previous step, SDBN proposes to refine its entry values with the following equation:

$$\tilde{\mathbf{A}}_i(p,q) = \begin{cases} 1 & \text{for } v_p, v_q \in M_k, \exists M_k \in \mathcal{M}; \\ \epsilon & \text{otherwise.} \end{cases} \tag{9}$$

  In the equation, term ϵ denotes a small constant.

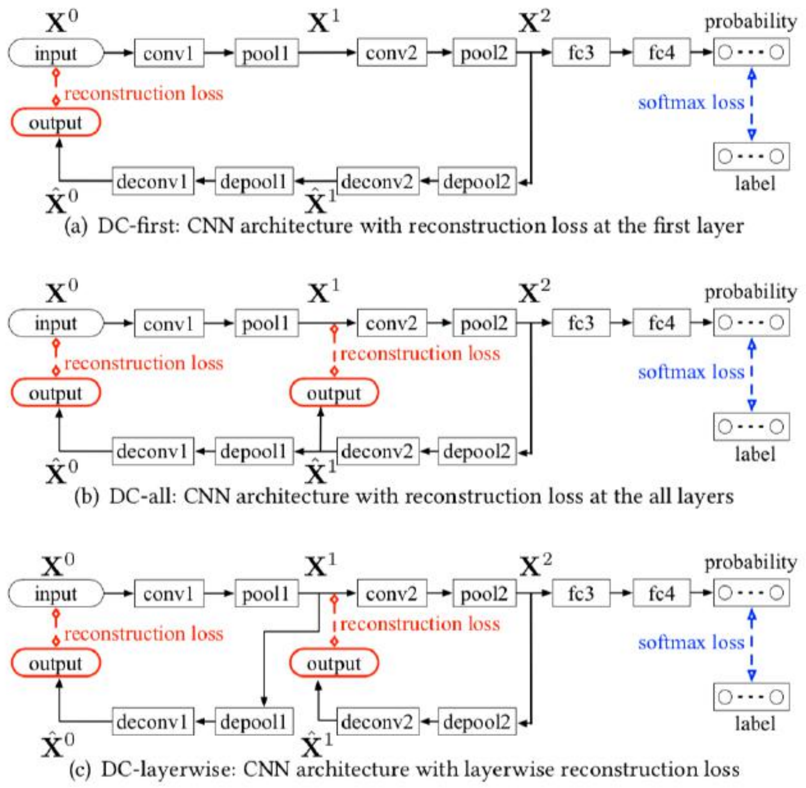
- **Module Identification Channel Creation**: From the reordered adjacency matrix $\tilde{\mathbf{A}}_i$ for graph $G_i \in \mathcal{G}$, the learned module identity information is actually not preserved. To effectively incorporate such information in the model, SDBN proposes to create one more channel $\mathbf{M}_i \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ for graph G_i, whose entry values can be denoted as follows:

$$\mathbf{M}_i(p,q) = \begin{cases} k & \text{for } v_p, v_q \in M_k, \exists M_k \in \mathcal{M}; \\ 0 & \text{otherwise.} \end{cases} \tag{10}$$

Formally, based on the above operations, the inputs for the representation learning component on graph G_i will be $\mathbf{X}_i = \left[\tilde{\mathbf{A}}_i; \mathbf{M}_i\right]$, which encodes much more information and can help learn more useful representations.

2.2.3 Learning of the SDBN Model

As illustrated in Figure 3, based on the input matrix $\mathbf{X}$ for the graphs in $\mathcal{G}$ (here, the subscript of $\mathbf{X}$ is not indicated and it can represent any graphs in $\mathcal{G}$), SDBN proposes to apply the convolutional neural network for the graph representation learning. To be



$\mathbf{X}^0$
$\mathbf{X}^1$
$\mathbf{X}^2$
probability
input
conv1
pool1
conv2
pool2
fc3
fc4
reconstruction loss
output
softmax loss
label
deconv1
depool1
deconv2
depool2
$\hat{\mathbf{X}}^0$
$\hat{\mathbf{X}}^1$
(a) DC-first: CNN architecture with reconstruction loss at the first layer
(b) DC-all: CNN architecture with reconstruction loss at the all layers
(c) DC-layerwise: CNN architecture with layerwise reconstruction loss

Figure 3: SDBN architecture with three types of unsupervised learning augmentation [7].

specific, the convolutional neural network used in SDBN involves two operators: *conv* and *pool*, which can be stacked together multiple times to form a deep architecture.

Formally, according to Figure 3, the intermediate representations of the input graphs as well as the corresponding labels in the SDBN can be represented with the following equations:

$$\begin{cases} \mathbf{X}^{(1)} & = pool\left(conv\left(\mathbf{X}; \boldsymbol{\Theta}\right)\right); \\ \mathbf{X}^{(2)} & = pool\left(conv\left(\mathbf{X}^{(1)}; \boldsymbol{\Theta}\right)\right); \\ \mathbf{x} & = reshape(\mathbf{X}^{(2)}); \\ \hat{\mathbf{y}} & = FC\left(\mathbf{x}; \boldsymbol{\Theta}\right), \end{cases} \tag{11}$$

where $reshape(\cdot)$ flattens the matrix to a matrix and $FC(\cdot)$ denotes the fully-connected layers in the model. In the above equations, $\boldsymbol{\Theta}$ denotes the involved variables in the model, which will be optimized.

Based on the above model, for all the graph instances $G_i \in \mathcal{G}$, we can represent the introduced loss terms by the model as

$$\ell(\boldsymbol{\Theta}) = \sum_{G_i \in \mathcal{G}} \ell(G_i; \boldsymbol{\Theta}) = \sum_{G_i \in \mathcal{G}} \ell(\mathbf{y}_i, \hat{\mathbf{y}}_i), \tag{12}$$

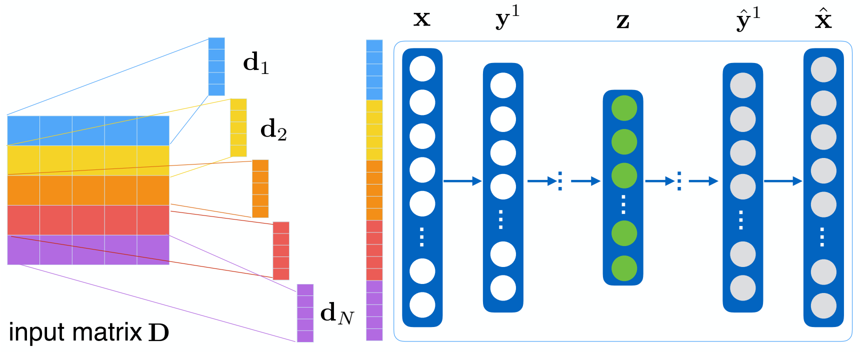


Figure 4: The LF&ER Framework for Latent Feature Extraction [6].

where $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$ represent the ground-truth label vector and the inferred label vector of graph G_i, respectively.

Meanwhile, in addition to the above loss term, SDBN also incorporates the autoencoder into the model learning process via the *depool* and *deconv* operations. The *conv* and *pool* operators mentioned above compose the encoder part, whereas the *deconv* and *depool* operators will form the decoder part. Formally, based on the learned intermediate representation $\mathbf{X}^{(2)}$ of the input graph matrix $\mathbf{X}$, SDBN computes the recovered representations as follows:

$$\begin{aligned}\hat{\mathbf{X}}^{(1)} &= deconv\left(depool\left(\mathbf{X}^{(2)}\right); \mathbf{\Theta}\right);\\ \hat{\mathbf{X}} &= deconv\left(depool\left(\hat{\mathbf{X}}^{(1)}\right); \mathbf{\Theta}\right).\end{aligned} \tag{13}$$

By minimizing the difference between $\hat{\mathbf{X}}^{(1)}$ and $\mathbf{X}^{(1)}$, as well as the difference between $\hat{\mathbf{X}}$ and $\mathbf{X}$, i.e.,

$$\ell(\hat{\mathbf{X}}^{(1)}, \mathbf{X}^{(1)}) = \left\|\hat{\mathbf{X}}^{(1)} - \mathbf{X}^{(1)}\right\|_2^2; \text{ and } \ell(\hat{\mathbf{X}}, \mathbf{X}) = \left\|\hat{\mathbf{X}} - \mathbf{X}\right\|_2^2 \tag{14}$$

SDBN can effectively learn the involved variables in the model. As illustrated in Figure 3, the decoder step can work in different manner, which will lead to different regularization terms on the intermediate representations. The performance comparison between IsoNN and SDBN is also reported in [4], and the readers may refer to [4, 7] for more detailed information of the models and the experimental evaluation results.

2.3 LF&ER: Deep Autoencoder based Latent Feature Extraction

Deep Autoencoder based Latent Feature Extraction (LF&ER) initially proposed in [6] serves an a latent feature extraction component in the final model introduced in that paper. Based on the input community real-estate POI graphs, LF&ER aims to learn the latent representations of the POI graphs, which will be used to infer the community vibrancy scores.

2.3.1 Deep Autoencoder Model

The LF&ER model works based on the deep autoencoder actually. Autoencoder is an unsupervised neural network model, which projects the instances in original feature representations into a lower-dimensional feature space via a series of non-linear mappings. Figure 4 shows that autoencoder model involves two steps: encode and decode. The encode part projects the original feature vector to the objective feature space, while the decode step recovers the latent feature representation to a reconstruction space. In autoencoder model, we generally need to ensure that the original feature representation of instances should be as similar to the reconstructed feature representation as possible.

Formally, let $\mathbf{x}$ represent the original feature representation of instance i, and $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \cdots, \mathbf{y}^{(o)}$ be the latent feature representations of the instance at hidden layers $1, 2, \cdots, o$ in the encode step respectively, the encoding result in the objective lower-dimension feature space can be represented as $\mathbf{z} \in \mathbb{R}^{d_z}$ with dimension d_z. Formally, the relationship between these vector variables can be represented with the following equations:

$$\begin{cases} \mathbf{y}^{(1)} &= \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}), \\ \mathbf{y}^{(k)} &= \sigma(\mathbf{W}^{(k)}\mathbf{y}^{(k-1)} + \mathbf{b}^{(k)}), \forall k \in \{2, 3, \cdots, o\}, \\ \mathbf{z} &= \sigma(\mathbf{W}^{(o+1)}\mathbf{y}^{(o)} + \mathbf{b}^{(o+1)}). \end{cases} \tag{15}$$

Meanwhile, in the decode step, the input will be the latent feature vector $\mathbf{z}$ (i.e., the output of the encode step), and the final output will be the reconstructed vector $\hat{\mathbf{x}}$. The latent feature vectors at each hidden layers can be represented as $\hat{\mathbf{y}}^{(o)}, \hat{\mathbf{y}}^{(o-1)}, \cdots, \hat{\mathbf{y}}^{(1)}$. The relationship between these vector variables can be denoted as

$$\begin{cases} \hat{\mathbf{y}}^{(o)} &= \sigma(\hat{\mathbf{W}}^{(o+1)}\mathbf{z} + \hat{\mathbf{b}}^{(o+1)}), \\ \hat{\mathbf{y}}^{(k-1)} &= \sigma(\hat{\mathbf{W}}^{(k)}\hat{\mathbf{y}}^{(k)} + \hat{\mathbf{b}}^{(k)}), \forall k \in \{2, 3, \cdots, o\}, \\ \hat{\mathbf{x}} &= \sigma(\hat{\mathbf{W}}^{(1)}\hat{\mathbf{y}}^{(1)} + \hat{\mathbf{b}}^{(1)}). \end{cases} \tag{16}$$

In the above equations, $\mathbf{W}$ and $\mathbf{b}$ with different subscripts denote the weight matrices and bias terms to be learned in the model. The objective of the autoencoder model is to minimize the loss between the original feature vector $\mathbf{x}$ and the reconstructed feature vector $\hat{\mathbf{x}}$. Formally, the loss term can be represented as

$$\ell(\mathbf{\Theta}) = \ell(\mathbf{x}, \hat{\mathbf{x}}; \mathbf{\Theta}) = \left\|\mathbf{x} - \hat{\mathbf{x}}\right\|_2^2, \tag{17}$$

where $\mathbf{\Theta}$ denotes the variables involved in the autoencoder model.

2.3.2 Latent Representation Learning

LF&ER proposes to learn the community allocation information for the vibrancy inference and ranking. Formally, *spatial structure* denotes the distribution of POIs inside the community, e.g., a grocery store lies between two residential buildings; a school is next to the police office. The *Spatial structure* can hardly be represented with explicit features extracted before, and LF&ER proposes to represent them with a set of latent feature vectors extracted from the *geographic distance graph* and the *mobility connectivity graph* defined in the previous subsection. The autoencoder model is applied here for the latent feature extraction.

Autoencoder model has been applied to embed the graph data into lower-dimensional spaces in many of the research works, which will obtain a latent feature representation for the nodes inside the graph. Different from these works, instead of calculating the latent feature for the POI categories inside the communities, LF&ER aims at obtaining the latent feature vector for the whole community, i.e., embedding the graph as one latent feature vector.

As shown in Figure 4, LF&ER transforms the matrix of the *graphical distance graph* (involving the POI categories) $\mathbf{D}$ into a vector, which can be denoted as

$$\mathbf{d} = reshape(\mathbf{D}) \in \mathbb{R}^{|\mathcal{V}|^2 \times 1}. \tag{18}$$

Vector $\mathbf{d}$ will be used as the input feeding into the autoencoder model. The latent embedding feature vector of $\mathbf{d}$ can be represented as $\mathbf{z}_D$ (i.e., the vector $\mathbf{z}$ as introduced in the autoencoder model in the previous section), which depicts the layout information of POI categories in the community in terms of the geographical distance. Besides the static layout based on *geographic distance graph*, the spatial structure of the POIs in the communities can also be revealed indirectly through the human mobility. For a pair of POI categories which are far away geographically, if people like to go between them frequently, it can display another type of structure of the POIs in terms of their functional correlations. Via the multiple fully connected layers, LF&ER will project such learned features to the objective vibrancy scores of the community. We will not introduce the model learning part here, since it also involves the ranking models and explicit feature engineering works, which is not closely related to the topic of this paper. The readers may refer to [6] for detailed description about the model and its learning process. In addition, autoencoder (i.e., the base model of LF&ER) is also compared against IsoNN, whose results are reported in [4].

3. Graph Neural Networks for Giant Networks

In this section, we will introduce the graph neural networks proposed for the representation learning tasks on giant networks instead. Formally, we can represent the giant network instance to be studied in this section as $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ and $\mathcal{E}$ denote the sets of nodes and links in the network, respectively. Different from the small graph data studied in Section 2, the nodes in the giant network G are partially annotated with labels instead. Formally, we can represent the set of labeled nodes as $\mathcal{V}_{\text{L}} = \{(v_{\text{L},1}, \mathbf{y}_{\text{L},1}), (v_{\text{L},2}, \mathbf{y}_{\text{L},2}), \cdots, (v_{\text{L},m}, \mathbf{y}_{\text{L},m})\}$, where $v_{\text{L},i} \in \mathcal{V}, \forall i \in \{1, 2, \cdots, m\}$ and $\mathbf{y}_{\text{L},i}$ denotes its label vector; whereas the remaining unlabeled nodes can be represented as $\mathcal{V}_{\text{U}} = \mathcal{V} \setminus \mathcal{V}_{\text{L}}$. In the case where all the involved network nodes are labeled, we will have $\mathcal{V}_{\text{L}} = \mathcal{V}$ and $\mathcal{V}_{\text{U}} = \emptyset$, which will be a special learning scenario of the general partial-labeled learning setting as studied in this paper. The giant network oriented graph neural networks aim at learning a mapping, i.e., $f : \mathcal{V} \to \mathbb{R}^{d_h}$, to obtain the feature vector representations of the nodes in the network, which can be utilized to infer their labels. To be more specific, the models to be introduced in this section include GCN [3], GAT [5], DifNN [8], GNL [1], GraphSage [2] and seGEN [9].

3.1 GCN: Graph Convolutional Network

Graph convolutional network (GCN) initially proposed in [3] introduces a *spectral graph convolution* operator for the graph data representation learning, which also provides several different approximations of the operator to encode both the graph structure and features of the nodes. GCN works well for the partially labeled giant networks, and the learned node representations can be effectively applied for the node classification task.

3.1.1 Spectral Graph Convolution

Formally, given an input network $G = (\mathcal{V}, \mathcal{E})$, its network structure information can be denoted as an adjacency matrix $\mathbf{A} = \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$. The corresponding normalized graph Laplacian matrix can be denoted as $\mathbf{L} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^\top$, where $\mathbf{D}$ is a diagonal matrix with entries $\mathbf{D}(i,i) = \sum_j \mathbf{A}(i,j)$ on its diagonal and $\mathbf{I} = diag(\mathbf{1}) \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{V}|}$ is an identity matrix with ones on its diagonal. The eigen-decomposition of matrix $\mathbf{L}$ can be denoted as $\mathbf{L} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^\top$, where $\mathbf{U}$ denotes the eigen-vector matrix and diagonal matrix $\mathbf{\Lambda}$ has eigen-values on its diagonal.

The *spectral convolution* operator defined on network G in GCN is denoted as a multiplication of an input signal vector $\mathbf{x} \in \mathbb{R}^{d_x}$ with a filter $\mathbf{g}_{\boldsymbol{\theta}} = diag(\boldsymbol{\theta})$ (parameterized by variable vector $\boldsymbol{\theta} \in \mathbb{R}^{d_\theta}$) in the Fourier domain as follows:

$$\mathbf{g}_{\boldsymbol{\theta}} * \mathbf{x} = \mathbf{U} \mathbf{g}_{\boldsymbol{\theta}} \mathbf{U}^\top \mathbf{x}, \tag{19}$$

where notation $\mathbf{U}^\top \mathbf{x}$ is defined as the graph Fourier transform of $\mathbf{x}$ and $\mathbf{g}_{\boldsymbol{\theta}}$ can be understood as a function on the eigen-values, i.e., $\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{\Lambda})$.

According to Equ. (19), the computation cost of the term on the right-hand-side will be $\mathcal{O}(|\mathcal{V}|^2)$. For the giant networks involving millions even billions of nodes, the computation of the graph convolution term will become infeasible, not to mention the eigen-decomposition of the Laplacian matrix $\mathbf{L}$ defined before. Therefore, to resolve such a problem, [3] introduces an approximation of the filter function $\mathbf{g}_{\boldsymbol{\theta}}(\mathbf{\Lambda})$ by a truncated expansion in terms of the Chebyshev polynomial $T_k(\cdot)$ up to the K_{th} order as follows:

$$\mathbf{g}_{\boldsymbol{\theta}} * \mathbf{x} \approx \sum_{k=0}^{K} \boldsymbol{\theta}(k) T_k(\tilde{\mathbf{L}}) \mathbf{x}, \tag{20}$$

where $\tilde{\mathbf{L}} = \frac{2}{\lambda_{max}} \mathbf{L} - \mathbf{I}$ and λ_{max} is the largest eigen-value in matrix $\mathbf{\Lambda}$. Vector $\boldsymbol{\theta} \in \mathbb{R}^k$ is a vector of Chebyshev coefficients. Noticing that the computational complexity of the term on the right-hand-side is $\mathcal{O}(|\mathcal{E}|)$, i.e., linear in terms of the edge numbers, which will be lower than that of Equ. (19) introduced before.

3.1.2 Graph Convolution Approximation

As proposed in [3], Equ. (20) can be further simplified by setting $K = 1$, $\lambda_{max} = 2$, $\boldsymbol{\theta}(0) = -\boldsymbol{\theta}(1) = \theta$, which will reduce the right hand-term of Equ. (20) approximately as follows:

$$\begin{aligned} \mathbf{g}_{\boldsymbol{\theta}} * \mathbf{x} &\approx \boldsymbol{\theta}(0)\mathbf{x} + \boldsymbol{\theta}(1)\tilde{\mathbf{L}}\mathbf{x} = \theta(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}})\mathbf{x} \\ &= \theta(\tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}})\mathbf{x}, \end{aligned} \tag{21}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\tilde{\mathbf{D}}$ is the diagonal matrix defined on $\tilde{\mathbf{A}}$ instead.

As illustrated in Figure 5, in the case when there exist C input channels, i.e., the input will be a matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times C}$, and F different filters defined above, the learned graph

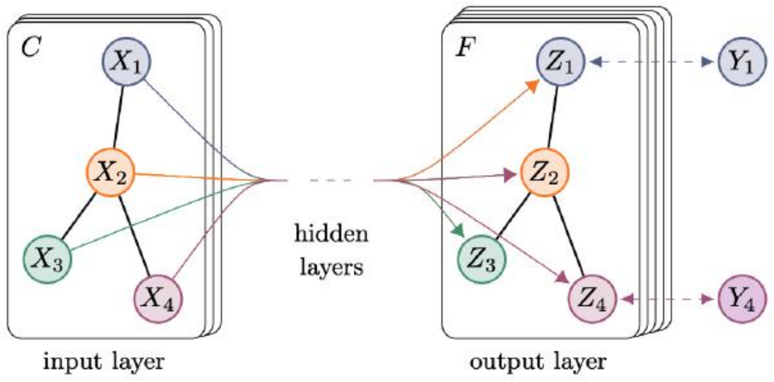


Figure 5: Schematic depiction of multi-layer GCN for semisupervised learning with C input channels and F feature maps in the output layer [3].

convolution feature representations will be

$$\begin{aligned}\mathbf{Z} &= (\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}})\mathbf{X}\mathbf{W},\\ &= \hat{\mathbf{A}}\mathbf{X}\mathbf{W}.\end{aligned} \tag{22}$$

where matrix $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$ can be pre-computed in advance. Matrix $\mathbf{W} \in \mathbb{R}^{C \times F}$ is the filter parameter matrix and $\mathbf{Z} \in \mathbb{R}^{|\mathcal{V}| \times F}$ will be the learned convolved representations of all the nodes. The computational time complexity of the operation will be $\mathcal{O}(|\mathcal{E}|FC)$.

3.1.3 Deep Graph Convolutional Network Learning

The GCN model can have a deeper architecture by involving multiple graph convolution operators defined in the previous sections. For instance, the GCN model with two layers can be represented with the following equations:

$$\begin{cases}\text{Layer 1: } \mathbf{Z} = \text{ReLU}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}_1\right); \\ \text{Output Layer: } \hat{\mathbf{Y}} = \text{softmax}\left(\hat{\mathbf{A}}\mathbf{Z}\mathbf{W}_2\right).\end{cases} \Rightarrow \hat{\mathbf{Y}} = \text{softmax}\left(\hat{\mathbf{A}}\text{ReLU}\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}_1\right)\mathbf{W}_2\right). \tag{23}$$

In the above equation, matrices $\mathbf{W}_1$ and $\mathbf{W}_2$ are the involved variables in the model. ReLU is used as the activation function for the hidden layer 1, and softmax function is used for the output result normalization. By comparing the inferred labels, i.e., $\hat{\mathbf{Y}}$, of the labeled instances against their ground-truth labels, i.e., $\mathbf{Y}$, the model variables can be effectively learned by minimizing the following loss function:

$$\ell(\mathbf{\Theta}) = -\sum_{v_i \in \mathcal{V}_L} \sum_{j=1}^{d_y} \mathbf{Y}(i,j) \log \hat{\mathbf{Y}}(i,j), \tag{24}$$

where $\mathbf{\Theta}$ covers all the variables in the model.

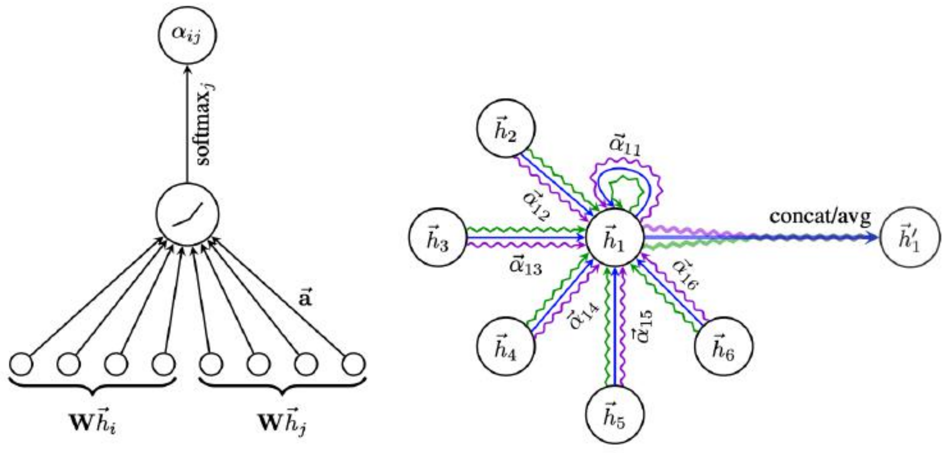


Figure 6: Schematic depiction of GAT with multi-head attention for the node representation update [5].

For representation simplicity, node subscript is used as its corresponding index in the label matrix $\mathbf{Y}$. Notation d_y denotes the number of labels in the studied problem, and $d_y = 2$ for the traditional binary classification tasks. The readers can also refer to [3] for detailed information of the GCN model.

3.2 GAT: Graph Attention Network

Graph attention network (GAT) initially proposed in [5] can be viewed as an extension of GCN. In updating the nodes' representations, instead of assigning the neighbors with fixed weights, i.e., values in matrix $\hat{\mathbf{A}}$ in Equ. (22) and Equ. (23), GAT introduces an attention mechanism to compute the weights based on the representations of the target node as well as its neighbors.

3.2.1 Graph Attention Coefficient Computation

Formally, given an input network $G = (\mathcal{V}, \mathcal{E})$ and the raw features of the nodes, the node features can be denoted as a matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d_x}$, where d_x denotes the dimension of the node feature vectors. Furthermore, for node $v_i \in \mathcal{V}$, its feature vector can also be represented as $\mathbf{x}_i = \mathbf{X}(i, :)$ for simplicity. Without considerations about the network structures, via a mapping matrix $\mathbf{W} \in \mathbb{R}^{d_x \times d_h}$, the nodes can be projected to their representations in the hidden layer. Meanwhile, to further incorporate the network structure information into the model, based on the network structure, the neighbor set of node v_i can be denoted as $\Gamma(v_i) = \{v_j | v_j \in \mathcal{V} \wedge (v_i, v_j) \in \mathcal{E}\} \cup \{v_i\}$, where $\{v_i\}$ is also added and treated as the *self-neighbor*. As illustrated in Figure 6, GAT proposes to compute the attention coefficient between nodes v_i and v_j (if $v_j \in \Gamma(v_i)$) as follows:

$$e_{i,j} = \text{LeakyReLU}\left(\mathbf{a}^\top \left(\mathbf{W}\mathbf{x}_i \sqcup \mathbf{W}\mathbf{x}_j\right)\right), \tag{25}$$

where $\mathbf{a} \in \mathbb{R}^{2d_h}$ is a variable vector for weighted sum of the entries in vector $\mathbf{W}\mathbf{x}_i \sqcup \mathbf{W}\mathbf{x}_j$ and $\sqcup$ denotes the concatenation operator of two vectors. LeakyReLU function is added here mainly for the model learning considerations.

To further normalize the coefficients among all the neighbors, GAT further adopts the softmax function based on the coefficients defined above. Formally, the final computed weight between nodes v_i and v_j can be denoted as

$$\begin{aligned}\alpha_{i,j} &= softmax(e_{i,j}) \\ &= \frac{\exp(e_{i,j})}{\sum_{v_k \in \Gamma(v_i)} \exp(e_{i,k})} \\ &= \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top\left(\mathbf{W}\mathbf{x}_i \sqcup \mathbf{W}\mathbf{x}_j\right)\right)\right)}{\sum_{v_k \in \Gamma(v_i)} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top\left(\mathbf{W}\mathbf{x}_i \sqcup \mathbf{W}\mathbf{x}_k\right)\right)\right)}.\end{aligned} \tag{26}$$

3.2.2 Representation Update via Neighborhood Aggregation

GAT effectively update the nodes' representations by aggregating the information from their neighbors (including the *self-neighbor*). Formally, the learned hidden representation of node v_i can be represented as

$$\mathbf{h}_i = \sigma\left(\sum_{v_j \in \Gamma(v_i)} \alpha_{i,j} \mathbf{W}\mathbf{x}_j\right). \tag{27}$$

GAT can be in a deeper architecture by involving multiple attentive node representation updating. In the deep architecture, for the upper layers, the representation vector $\mathbf{h}_i$ will be treated as the inputs feature vector instead, and we will not over-elaborate that here.

3.2.3 Multi-Head Attention Aggregation

As introduced in [5], to stabilize the learning process of the model, GAT can be further extended to include the multi-head attention as illustrated in Figure 6. Specifically, let K denote the number of involved attention mechanisms. Based on each attention mechanism, the learned representations of node v_i based on the above descriptions (i.e., Equ. (27)) can be denoted as $\mathbf{h}_i^{(1)}$, $\mathbf{h}_i^{(2)}$, $\cdots$, $\mathbf{h}_i^{(K)}$, respectively. By further aggregating such learned representations together, the ultimate learned representation of node v_i can be denoted as

$$\mathbf{h}'_i = \text{Aggregate}(\mathbf{h}_i^{(1)}, \mathbf{h}_i^{(2)}, \cdots, \mathbf{h}_i^{(K)}). \tag{28}$$

Several different aggregation function is tried in [5], including *concatenation* and *average*:

- *Concatenation*:

$$\mathbf{h}'_i = \bigsqcup_{k=1}^{K} \sigma\left(\sum_{v_j \in \Gamma(v_i)} \alpha_{i,j}^{(k)} \mathbf{W}^{(k)} \mathbf{x}_j\right). \tag{29}$$

- *Average*:

$$\mathbf{h}'_i = \sigma\left(\frac{1}{K} \sum_{k=1}^{K} \sum_{v_j \in \Gamma(v_i)} \alpha_{i,j}^{(k)} \mathbf{W}^{(k)} \mathbf{x}_j\right). \tag{30}$$

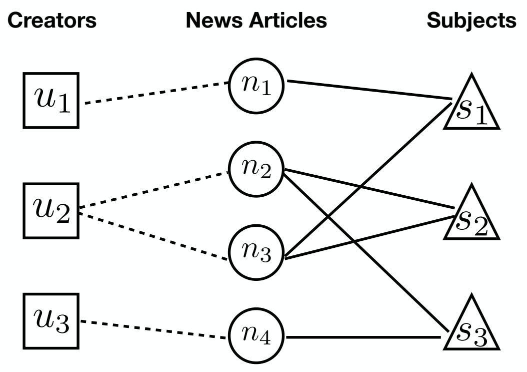


Figure 7: An example of the news augmented heterogeneous social network

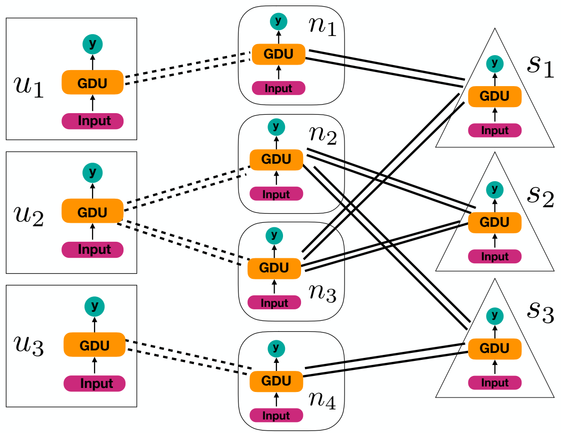


Figure 8: The DifNN model architecture build for the input heterogeneous social network

The learning process of the GAT model is very similar to that of GCN introduced in Section 3.1.3, and we will not introduce that part again here. The readers can also refer to [5] for more detailed descriptions about the model as well as its experimental performance.

3.3 DifNN: Deep Diffusive Neural Network

Deep diffusive network (DifNN) model initially introduced in [8] aims at modeling the diverse connections in heterogeneous information networks, which contains multiple types of nodes and links. DifNN is based on a new type of neuron, namely *gated diffusive unit* (GDU), which can be extended to incorporate the inputs from various groups of neighbors.

3.3.1 Model Architecture

Given a heterogeneous input network $G = (\mathcal{V}, \mathcal{E})$, the node set $\mathcal{V}$ in the network can be divided into multiple subsets depending on their node types. It is similar for the links in set $\mathcal{E}$. Here, for the representation simplicity, we will follow the news augmented heterogeneous social network example illustrated in [8] when introducing the model. As illustrated in Figure 7, there exist three different types of nodes (i.e., *creator*, *news article* and *subject*) and two different types of links (i.e., the *creator-article link* and *article-subject link*) in the network. Formally, the node set can be categories into three subsets, i.e., $\mathcal{V} = \mathcal{U} \cup \mathcal{N} \cup \mathcal{S}$, and the link set can be categorized into two subsets, i.e., $\mathcal{E} = \mathcal{E}_{u,n} \cup \mathcal{E}_{n,s}$.

For each node in the network, e.g., $v_i \in \mathcal{V}$, its extracted raw feature vector can be denoted as $\mathbf{x}_i$. As introduced at the beginning of Section 3, in many cases, the network is partially labeled. Formally, the label vector of node v_i is represented as $\mathbf{y}_i$. For each nodes in the input network, DifNN utilizes one GDU (which will be introduced in the following subsection) to model its representations and the connections with other neighboring nodes. For instance, based on the input network in Figure 7, its corresponding DifNN model architecture can be represented in Figure 8. Via the gdu neuron unit, DifNN can effectively project the

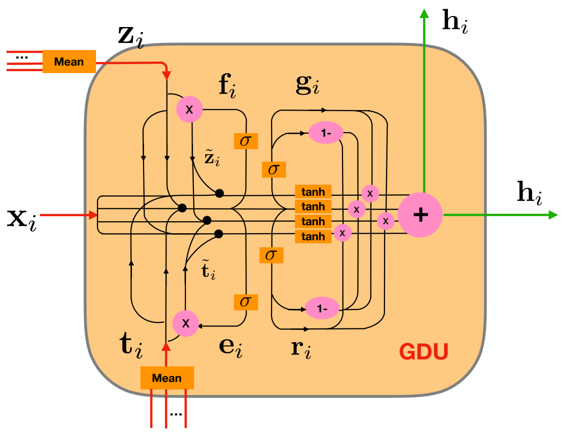


Figure 9: An illustration of the gated diffusive unit (GDU).

node inputs to their corresponding labels. The parameters involved in the DIFNN model can be effectively trained based on the labeled nodes via the back propagation algorithm. In the following two subsections, we will introduce the detailed information about GDU as well as the DIFNN model training.

3.3.2 Gated Diffusive Unit

To introduce the GDU neuron, we can take news article nodes as an example here. Formally, among all the inputs of the GDU model, $\mathbf{x}_i$ denotes the extracted feature vector for news articles, $\mathbf{z}_i$ represents the input from other GDUs corresponding to subjects, and $\mathbf{t}_i$ represents the input from other GDUs about creators. Considering that the GDU for each news article may be connected to multiple GDUs of subjects and creators, the *mean*$(\cdot)$ of the outputs from the GDUs corresponding to these subjects and creators will be computed as the inputs $\mathbf{z}_i$ and $\mathbf{t}_i$ instead respectively, which is also indicated by the GDU architecture illustrated in Figure 9. For the inputs from the subjects, GDU has a gate called the "forget gate", which may update some content of $\mathbf{z}_i$ to forget. The forget gate is important, since in the real world, different news articles may focus on different aspects about the subjects and "forgetting" part of the input from the subjects is necessary in modeling. Formally, the "forget gate" together with the updated input can be represented as

$$\tilde{\mathbf{z}}i = \mathbf{f}_i \otimes \mathbf{z}_i, \text{ where } \mathbf{f}_i = \sigma\left(\mathbf{W}_f \left[\mathbf{x}_i^\top, \mathbf{z}_i^\top, \mathbf{t}_i^\top\right]^\top\right). \tag{31}$$

Here, operator $\otimes$ denotes the entry-wise product of vectors and $\mathbf{W}_f$ represents the variable of the forget gate in GDU.

Meanwhile, for the input from the creator nodes, a new node-type "adjust gate" is introduced in GDU. Here, the term "adjust" models the necessary changes of information between different node categories (e.g., from creators to articles). Formally, the "adjust gate" as well as the updated input can be denoted as

$$\tilde{\mathbf{t}}_i = \mathbf{e}_i \otimes \mathbf{t}_i, \text{ where } \mathbf{e}_i = \sigma\left(\mathbf{W}_e \left[\mathbf{x}_i^\top, \mathbf{z}_i^\top, \mathbf{t}_i^\top\right]^\top\right), \tag{32}$$

where $\mathbf{W}_e$ denotes the variable matrix in the adjust gate.

GDU allows different combinations of these input/state vectors, which are controlled by the selection gates $\mathbf{g}_i$ and $\mathbf{r}_i$ respectively. Formally, the final output of GDU will be

$$\begin{aligned}\mathbf{h}_i &= \mathbf{g}_i \otimes \mathbf{r}_i \otimes \tanh\left(\mathbf{W}_u[\mathbf{x}_i^\top, \tilde{\mathbf{z}}_i^\top, \tilde{\mathbf{t}}_i^\top]^\top\right) \\ &+ (\mathbf{1} - \mathbf{g}_i) \otimes \mathbf{r}_i \otimes \tanh\left(\mathbf{W}_u[\mathbf{x}_i^\top, \mathbf{z}_i^\top, \tilde{\mathbf{t}}_i^\top]^\top\right) \\ &+ \mathbf{g}_i \otimes (\mathbf{1} - \mathbf{r}_i) \otimes \tanh\left(\mathbf{W}_u[\mathbf{x}_i^\top, \tilde{\mathbf{z}}_i^\top, \mathbf{t}_i^\top]^\top\right) \\ &+ (\mathbf{1} - \mathbf{g}_i) \otimes (\mathbf{1} - \mathbf{r}_i) \otimes \tanh\left(\mathbf{W}_u[\mathbf{x}_i^\top, \mathbf{z}_i^\top, \mathbf{t}_i^\top]^\top\right),\end{aligned} \tag{33}$$

where $\mathbf{g}_i = \sigma(\mathbf{W}_g \left[\mathbf{x}_i^\top, \mathbf{z}_i^\top, \mathbf{t}_i^\top\right]^\top)$, and $\mathbf{r}_i = \sigma(\mathbf{W}_r \left[\mathbf{x}_i^\top, \mathbf{z}_i^\top, \mathbf{t}_i^\top\right]^\top)$, and term $\mathbf{1}$ denotes a vector filled with value 1. Operators $\oplus$ and $\ominus$ denote the entry-wise addition and minus operation of vectors. Matrices $\mathbf{W}_u$, $\mathbf{W}_g$, $\mathbf{W}_r$ represent the variables involved in the components. Vector $\mathbf{h}_i$ will be the output of the GDU model.

The introduced GDU model also works for both the news subjects and creator nodes in the network. When applying the GDU to model the states of the subject/creator nodes with two input only, the remaining input port can be assigned with a default value (usually vector $\mathbf{0}$). In the following section, we will introduce how to learn the parameters involved in the DifNN model for concurrent inference of multiple nodes.

3.3.3 DifNN Model Learning

In the DifNN model as shown in Figure 8, based on the output state vectors of news articles, news creators and news subjects, the framework will project the feature vectors to their labels. Formally, given the state vectors $\mathbf{h}_{n,i}$ of news article n_i, $\mathbf{h}_{u,j}$ of news creator u_j, and $\mathbf{h}_{s,l}$ of news subject s_l, their inferred labels can be denoted as vectors $\mathbf{y}_{n,i}, \mathbf{y}_{u,j}, \mathbf{y}_{s,l} \in \mathcal{R}^{|\mathcal{Y}|}$ respectively, which can be represented as

$$\begin{cases}\mathbf{y}_{n,i} &= softmax\left(\mathbf{W}_n \mathbf{h}_{n,i}\right), \\ \mathbf{y}_{u,j} &= softmax\left(\mathbf{W}_u \mathbf{h}_{u,j}\right), \\ \mathbf{y}_{s,l} &= softmax\left(\mathbf{W}_s \mathbf{h}_{s,l}\right).\end{cases} \tag{34}$$

where $\mathbf{W}_u$, $\mathbf{W}_n$ and $\mathbf{W}_s$ define the weight variables projecting state vectors to the output vectors.

Meanwhile, based on the news articles in the training set $\mathcal{T}_n \subset \mathcal{N}$ with the ground-truth label vectors $\{\hat{\mathbf{y}}_{n,i}\}_{n_i \in \mathcal{T}_n}$, the loss function of the framework for news article label learning are defined as the cross-entropy between the prediction results and the ground truth:

$$\ell(\mathcal{T}_n; \mathbf{\Theta}) = -\sum_{n_i \in \mathcal{T}_n} \sum_{k=1}^{|\mathcal{Y}|} \hat{\mathbf{y}}_{n,i}(k) \log \mathbf{y}_{n,i}(k). \tag{35}$$

Similarly, the loss terms introduced by news creators and subjects based on training sets $\mathcal{T}_u \subset \mathcal{U}$ and $\mathcal{T}_s \subset \mathcal{S}$ can be denoted as

$$\ell(\mathcal{T}_u; \mathbf{\Theta}) = -\sum_{u_j \in \mathcal{T}_u} \sum_{k=1}^{|\mathcal{Y}|} \hat{\mathbf{y}}_{u,j}(k) \log \mathbf{y}_{u,j}(k), \tag{36}$$

$$\ell(\mathcal{T}_s; \mathbf{\Theta}) = - \sum_{s_l \in \mathcal{T}_s} \sum_{k=1}^{|\mathcal{Y}|} \hat{\mathbf{y}}_{s,l}(k) \log \mathbf{y}_{s,l}(k), \tag{37}$$

where $\mathbf{y}_{u,j}$ and $\hat{\mathbf{y}}_{u,j}$ (and $\mathbf{y}_{s,l}$ and $\hat{\mathbf{y}}_{s,l}$) denote the prediction result vector and ground-truth vector of creator (and subject) respectively.

Formally, the main objective function of the DifNN model can be represented as follows:

$$\min_{\mathbf{\Theta}} \ell(\mathcal{T}_n; \mathbf{\Theta}) + \ell(\mathcal{T}_u; \mathbf{\Theta}) + \ell(\mathcal{T}_s; \mathbf{\Theta}) + \alpha \cdot reg(\mathbf{\Theta}), \tag{38}$$

where $\mathbf{\Theta}$ denotes all the involved variables to be learned, term $reg(\mathbf{\Theta})$ represents the regularization term (i.e., the sum of L_2 norm on the variable vectors and matrices), and α denotes the regularization term weight. By resolving the optimization functions, variables in the model can be effectively learned with the *back-propagation* algorithm. For the news articles, creators and subjects in the testing set, their predicted labels will be outputted as the final result.

3.4 GNL: Graph Neural Lasso

Graph neural lasso (GNL) initially proposed in [1] is a graph neural regression model and it can effectively incorporate the historical time-series data of multiple instances for addressing the dynamic network regression problem. GNL extends the GDU neuron [8] (also introduced in Section 3.3.2) for incorporating both the network internal relationships and the network dynamic relationships between sequential network snapshots.

3.4.1 Dynamic Gated Diffusive Unit

GNL also adopts GDU as the basic neuron unit and extends it to the dynamic network regression problem settings, which can model both the network snapshot internal connections and the temporal dependency relationships between sequential network snapshots for the nodes.

Formally, given the time series data about a set of connected entities, such data can be represented as a dynamic network set $\mathcal{G} = \{G^{(1)}, G^{(2)}, \cdots, G^{(t)}\}$, where t denotes the maximum timestamp. For each network $G^{(\tau)} \in \mathcal{G}$, it can be denoted as $G^{(\tau)} = (\mathcal{V}^{(\tau)}, \mathcal{E}^{(\tau)})$ involving the node set $\mathcal{V}^{(\tau)}$ and link set $\mathcal{E}^{(\tau)}$, respectively. Given a node v_i in network $G^{(\tau)}$, its in-neighbors and out-neighbors in the network can be denoted as sets $\Gamma_{in}(v_i) = \{v_j | v_j \in \mathcal{V}^{(\tau)} \wedge (v_j, v_i) \in \mathcal{E}^{(\tau)}\}$ and $\Gamma_{out}(v_i) = \{v_j | v_j \in \mathcal{V}^{(\tau)} \wedge (v_i, v_j) \in \mathcal{E}^{(\tau)}\}$. Here, the link direction denotes the influences among the nodes. If the influences in the studied networks are bi-directional, the in/out neighbor sets will be identical, i.e., $\Gamma_{in}(v_i) = \Gamma_{out}(v_i)$.

For node v_i in network $G^{(\tau)}$ of the τ_{th} timestamp, the input attribute values of v_i can be denoted as an input feature vector $\mathbf{x}_i^{(\tau)} \in \mathbb{R}^{d_x}$. GDU maintains a hidden state vector for each node, and the vector of node v_i at timestamp τ can be denoted as $\mathbf{h}_i^{(\tau)} \in \mathbb{R}^{d_h}$. As illustrated in Figure 10, besides the feature vector $\mathbf{x}_i^{(\tau)}$ and hidden state vector $\mathbf{h}_i^{(\tau)}$ inputs, the GDU neuron of v_i will also accept the inputs from v_i's input neighbor nodes, i.e., $\{\mathbf{h}_j^{(\tau)}\}_{v_j \in \Gamma_{in}(v_i)}$, which will be integrated via certain aggregation operators:

$$\mathbf{z}_i^{(\tau)} = \text{Aggregate}\left(\{\mathbf{h}_j^{(\tau)}\}_{v_j \in \Gamma_{in}(v_i)}\right). \tag{39}$$

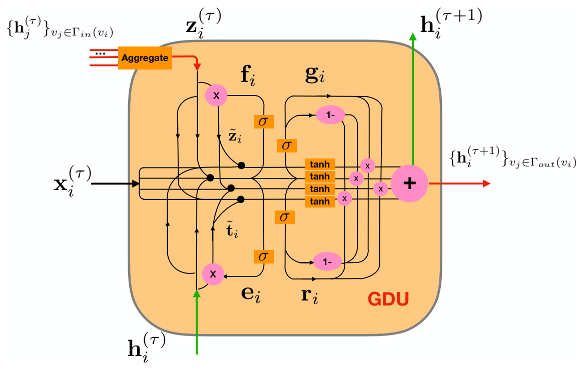


Figure 10: The detailed architecture of the GDU neuron of node v_i at timestamp τ.

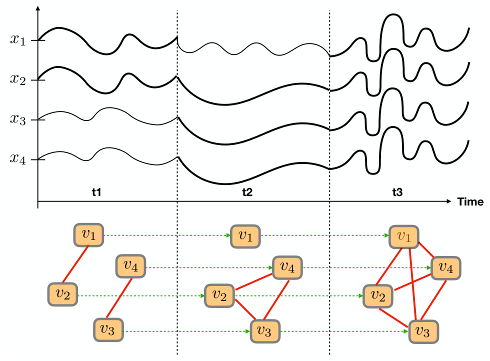


Figure 11: The framework outline of GNL based on GDU.

The Aggregate($\cdot$) operator used in GNL will be introduced in detail in the next subsection.

A common problem with existing graph neural network models is over-smoothing, which will reduce all the nodes in the network to similar hidden representations. Such a problem will be much more serious when the model involves a deep architecture with multiple layers. To resolve such a problem, besides the attention mechanism to be introduced later, GDU introduces several gates for the neural state adjustment as introduced in Section 3.3.2. Formally, based on the input vectors $\mathbf{x}_i^{(\tau)}$, $\mathbf{z}_i^{(\tau)}$ and $\mathbf{h}_i^{(\tau)}$, the representation of node v_i in the next timestamp $\tau + 1$ can be represented as

$$\mathbf{h}_i^{(\tau+1)} = \text{GDU}\left(\mathbf{x}_i^{(\tau)}, \mathbf{z}_i^{(\tau)}, \mathbf{h}_i^{(\tau)}; \mathbf{\Theta}\right). \tag{40}$$

The concrete representation of the $GDU(\cdot)$ function is similar to Equ. (31)-(33) introduced before, and $\mathbf{\Theta}$ denotes the variables involved in the GDU neuron.

3.4.2 Attentive Neighborhood Influence Aggregation

In this part, we will introduce the Aggregate($\cdot$) operator used in Equ. (39) for node neighborhood influence integration proposed in [1]. The GNL model defines such an operator based on an attention mechanism. Formally, given the node v_i and its in-neighbor set $\Gamma_{in}(v_i)$, for any node $v_j \in \Gamma_{in}(v_i)$, GNL quantifies the influence coefficient of v_j on v_i based on their hidden state vectors $\mathbf{h}_j^{(\tau)}$ and $\mathbf{h}_i^{(\tau)}$ as follows:

$$\alpha_{j,i}^{(\tau)} = \text{AttInf}(e_{j,i}^{(\tau)}) = \frac{\exp(e_{j,i}^{(\tau)})}{\sum_{v_k \in \Gamma_{out}(v_j)} \exp(e_{j,k}^{(\tau)})}, \text{ where } e_{j,i}^{(\tau)} = \text{Linear}(\mathbf{W}_a \mathbf{h}_j^{(\tau)} \sqcup \mathbf{W}_a \mathbf{h}_i^{(\tau)}; \mathbf{w}_a). \tag{41}$$

In the above equation, operator Linear($\cdot; \mathbf{w}_a$) denotes a linear sum of the input vector parameterized by weight vector $\mathbf{w}_a$. According to [5], out of the model learning concerns, the above influence coefficient term can be slightly changed by adding the LeakyReLU function into its definition. Formally, the final influence coefficient used in GNL is represented as follows:

$$\alpha_{j,i}^{(\tau)} = \text{AttInf}(\mathbf{h}_j^{(\tau)}, \mathbf{h}_i^{(\tau)}; \mathbf{W}_a, \mathbf{w}_a) = \frac{\exp(\text{LeakyReLU}(\text{Linear}(\mathbf{W}_a \mathbf{h}_j^{(\tau)} \sqcup \mathbf{W}_a \mathbf{h}_i^{(\tau)}; \mathbf{w}_a)))}{\sum_{v_k \in \Gamma_{out}(v_j)} \exp(\text{LeakyReLU}(\text{Linear}(\mathbf{W}_a \mathbf{h}_j^{(\tau)} \sqcup \mathbf{W}_a \mathbf{h}_k^{(\tau)}; \mathbf{w}_a)))}. \quad (42)$$

Considering that in our problem setting the links in the dynamic networks are unknown and to be inferred, the above influence coefficient term $\alpha_{j,i}$ actually quantifies the existence probability of the influence link (v_j, v_i), i.e., the inference results of the links. Furthermore, based on the influence coefficient, the concrete representation of Equ. (39) will be represented as follows:

$$\mathbf{z}_i^{(\tau)} = \text{Aggregate}\left(\{\mathbf{h}_j^{(\tau)}\}_{v_j \in \Gamma_{in}(v_i)}\right) = \sigma\left(\sum_{v_j \in \Gamma_{in}(v_i)} \alpha_{j,i}^{(\tau)} \mathbf{W}_a \mathbf{h}_j^{(\tau)}\right). \quad (43)$$

3.4.3 Graph Neural Lasso Model Learning

In this part, we will introduce the architecture of the GNL model together with its learning settings. Formally, given the input dynamic network set $\mathcal{G} = \{G^{(1)}, G^{(2)}, \cdots, G^{(t)}\}$, GNL shifts a window of size τ along the networks in the order of their timestamps. The network snapshots covered by the window, e.g., $G^{(k)}$, $G^{(k+1)}$, $\cdots$, $G^{(k+\tau-1)}$, will be taken as the model input of GNL to infer the network $G^{(k+\tau)}$ in following timestamp (where $k, k+1, \cdots, k+\tau \in \{1, 2, \cdots, t\}$). According to the above descriptions, the inferred attribute values of all the nodes and their potential influence links in network $G^{(k+\tau)}$ can be represented as

$$\begin{aligned} \hat{\mathbf{x}}_i^{(\tau+1)} &= \text{FC}(\mathbf{h}_i^{(\tau+1)}; \mathbf{\Theta}), \forall v_i \in \mathcal{V}^{(\tau+1)}; \\ \alpha_{j,i}^{(\tau)} &= \text{AttInf}(\mathbf{h}_j^{(\tau+1)}, \mathbf{h}_i^{(\tau+1)}; \mathbf{\Theta}), \forall v_i, v_j \in \mathcal{V}^{(\tau+1)}, \end{aligned} \quad (44)$$

In the above equation, term $\mathbf{h}_i^{(\tau+1)}$ is defined in Equ. (40) and $\mathbf{\Theta}$ covers all the involved variables used in the GNL model. By comparing the inferred node attribute values, e.g., $\hat{\mathbf{x}}_i^{(\tau+1)}$, against the ground truth values, e.g., $\mathbf{x}_i^{(\tau+1)}$, the quality of the inference results by GNL can be effectively measured with some loss functions, e.g., mean square error:

$$\ell(\mathbf{\Theta}) = \frac{1}{|\mathcal{V}^{(\tau+1)}|} \sum_{v_i \in \mathcal{V}^{(\tau+1)}} \ell(v_i; \mathbf{\Theta}) = \frac{1}{|\mathcal{V}^{(\tau+1)}|} \sum_{v_i \in \mathcal{V}^{(\tau+1)}} \left\| \hat{\mathbf{x}}_i^{(\tau+1)} - \mathbf{x}_i^{(\tau+1)} \right\|_2^2. \quad (45)$$

In addition, similar to Lasso, to avoid overfitting, GNL proposes to add a regularization term in the objective function to maintain the sparsity of the variables. Formally, the final objective function of the GNL model can be represented as follows:

$$\min_{\mathbf{\Theta}} \ell(\mathbf{\Theta}) + \beta \cdot \|\mathbf{\Theta}\|_1, \quad (46)$$

where term $\|\mathbf{\Theta}\|_1$ denotes the sum of the L_1-norm regularizer of all the involved variables in the model and β is the hyper-parameter weight of the regularization term. More information about the model learning as well as how to handle the non-derivable L_1-norm regularization term is available in [1].

Algorithm 1 Algorithm GraphSage

```
Require: Network $G = (\mathcal{V}, \mathcal{E})$; Input Node Feature: $\{\mathbf{x}_i\}_{v_i \in \mathcal{V}}$; Model Depth: $K$.
Ensure: Learned Representations $\{\mathbf{z}_i\}_{v_i \in \mathcal{V}}$.
 1: Initialize $\mathbf{h}_i^{(0)} = \mathbf{x}_i, \forall v_i \in \mathcal{V}$
 2: for $k \in \{1, 2, \cdots, K\}$ do
 3:     for $v_i \in \mathcal{V}$ do
 4:         $\mathcal{N}(v_i) = \text{Sample}\left(\Gamma(v_i)\right)$
 5:         $\mathbf{h}_{\Gamma(v_i)}^{(k)} = \text{Aggregate}\left(\left\{\mathbf{h}_j^{(k-1)} | v_j \in \mathcal{N}(v_i)\right\}\right)$
 6:         $\mathbf{h}_i^{(k)} = \sigma\left(\mathbf{W}^{(k)}\left(\mathbf{h}_{\Gamma(v_i)}^{(k)} \sqcup \mathbf{h}_i^{(k-1)}\right)\right)$
 7:     end for
 8:     $\mathbf{h}_i^{(k)} = \text{Normalize}\left(\mathbf{h}_i^{(k)}\right)$
 9: end for
10: $\mathbf{z}_i = \mathbf{h}_i^{(K)}, \forall v_i \in \mathcal{V}$
```

3.5 GraphSage: Graph Sample and Aggregate

GraphSage introduced in [2] is an inductive model which focuses on leveraging node feature information for effective network node embedding. Instead of training individual embedding for each node, GraphSage learns a function that generate embeddings by sampling and aggregating features from nearby neighbors.

3.5.1 Framework Descriptions

As illustrated in Algorithm 1, the GraphSage algorithm accepts the network structure $G = (\mathcal{V}, \mathcal{E})$, input raw feature vectors $\{\mathbf{x}_i\}_{v_i \in \mathcal{V}}$ and model depth K as the inputs, which will return the learned representations of nodes in the network as the output. Several functions will be called in the algorithm, including *Sample*$(\cdot)$, *Aggregate*$(\cdot)$ and *Normalize*$(\cdot)$.

The forward computational process of GraphSage works iteratively layers by layers and nodes by nodes. Formally, the representations of nodes at each layer can be represented as vectors $\{\mathbf{h}_i^{(k)}\}_{v_i \in \mathcal{V}, k \in \{1,2,\cdots,K\}}$, where $\mathbf{h}_i^{(k)}$ denotes the representation of v_i at the k_{th} layer. For all the nodes, their representations at layer 0 are initialized with their raw features, i.e., $\mathbf{h}_i^{(0)} = \mathbf{x}_i, \forall v_i \in \mathcal{V}$. Given a node v_i, its neighbors can be represented as set $\Gamma(v_i) = \{v_j | v_j \in \mathcal{V} \wedge (v_i, v_j) \in \mathcal{E}\}$. GraphSage calls an aggregation function to effectively aggregate the neighbors' representations. However, instead of directly working on the complete neighbor set $\Gamma(v_i)$, GraphSage proposes to sample a subset of the neighbors prior to information aggregation, which is denoted as

$$\mathcal{N}(v_i) = \text{Sample}\left(\Gamma(v_i)\right) \tag{47}$$

where $\mathcal{N}(v_i) \subset \Gamma(v_i)$ has a fixed size for all the nodes in the network and the sampling process follows a uniform distribution.

At the k_{th} layer, via aggregating all the representations of the nodes in $\mathcal{N}(v_i)$ from the $(k-1)_{th}$ layer, GraphSage defines the a pseudo-representation for v_i as follows:

$$\mathbf{h}_{\Gamma(v_i)}^{(k)} = \text{Aggregate}\left(\left\{\mathbf{h}_j^{(k-1)} | v_j \in \mathcal{N}(v_i)\right\}\right). \tag{48}$$

The concrete representations of the Aggregate $(\cdot)$ function will be introduced later.

By concatenating the computed pseudo-representation $\mathbf{h}^{(k)}_{\Gamma(v_i)}$ and its representation at the $(k-1)_{th}$ layer, GraphSage defines the node representation updating equation as follows:

$$\mathbf{h}^{(k)}_i = \sigma\left(\mathbf{W}^{(k)}\left(\mathbf{h}^{(k)}_{\Gamma(v_i)} \sqcup \mathbf{h}^{(k-1)}_i\right)\right), \tag{49}$$

$$\mathbf{h}^{(k)}_i = \text{Normalize}\left(\mathbf{h}^{(k)}_i\right) = \frac{\mathbf{h}^{(k)}_i}{\left\|\mathbf{h}^{(k)}_i\right\|_2}, \tag{50}$$

where operator $\sqcup$ denotes the concatenation of two vectors and GraphSage normalizes the vector $\mathbf{h}^{(k)}_i$ by dividing it with its modulus.

3.5.2 Aggregator Function

The "orderless" property of the the neighbor nodes poses more challenges on the aggregation operator to be used in GraphSage. Besides the aggregate function used above, several other aggregators can also be used for the information integration, which are provided as follows:

- *Mean Aggregator*: The mean aggregator is very similar to the propagation rules used in GCN introduced in Section 3.1, which replaces Equ. (48) and Equ. (49) with the following equation instead:

$$\mathbf{h}^{(k)}_i = \sigma\left(\mathbf{W}^{(k)}\text{Mean}\left(\left\{\mathbf{h}^{(k-1)}_i\right\} \cup \left\{\mathbf{h}^{(k-1)}_j | v_j \in \mathcal{N}(v_i)\right\}\right)\right). \tag{51}$$

- *LSTM Aggregator*: Much more complex aggregators, e.g., LSTM, can also be adopted for the nodes representation aggregation and updating in GraphSage. By randomly permuting the neighbors in set $\Gamma(v_i)$, LSTM can be applied on the unordered set, where the output of the last unit can be obtained as the output. In other words, Equ. (48) can be updated as follows:

$$\mathbf{h}^{(k)}_{\Gamma(v_i)} = \text{LSTM}\left(\left\{\mathbf{h}^{(k-1)}_j | v_j \in \mathcal{N}(v_i)\right\}\right) \tag{52}$$

- *Pooling Aggregator*: The pooling aggregator works with a max pooling layer to integrate the information from the neighbors, and Equ. (48) can be replaced as follows:

$$\mathbf{h}^{(k)}_{\Gamma(v_i)} = \max\left(\left\{\sigma\left(\mathbf{W}^{(k)}\mathbf{h}^{(k-1)}_j + \mathbf{b}^{(k)}\right) | v_j \in \mathcal{N}(v_i)\right\}\right) \tag{53}$$

3.6 seGEN: Sample and Ensemble Genetic Evolutionary Network

Sample-Ensemble Genetic Evolutionary Network (seGEN) first proposed in [9] can serve as an alternative approach to GNN models on giant networks. Instead of building one single graph neural network, based on a set of sampled sub- graphs, seGEN adopts a genetic-evolutionary learning strategy to build a group of unit models generations by generations. The unit models incorporated in seGEN can be either traditional graph representation learning models or the recent graph neural network models with a much "narrower" and "shallower" architecture. The learning results of each instance at the final generation will be effectively combined from each unit model via diffusive propagation and ensemble learning strategies.

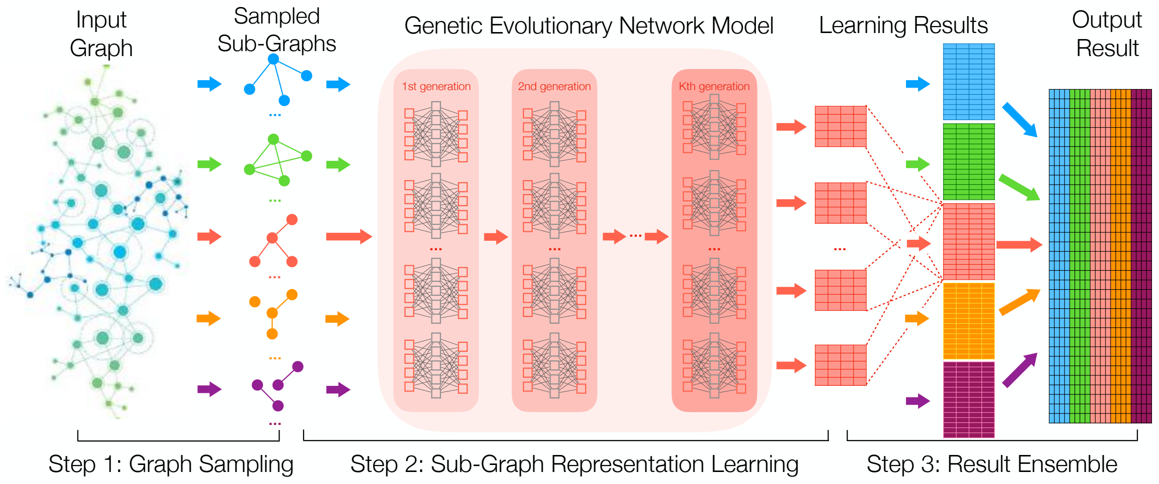


Figure 12: The SEGEN Framework [9] with Three Main Steps (Step 1: graph sampling to achieve a set of sub-graph instances; Step 2: sub-graph representation learning to get the representation features of nodes; Step 3: result ensemble of the learned node representations on each sub-graph by the unit models to get the final node representation).

3.6.1 SEGEN Architecture Description

In framework SEGEN, instead of handling the input large-scale graph data directly, it proposes to sample a subset (of set size s) of small-sized sub-graphs (of a pre-specified sub-graph size k) instead and learn the representation feature vectors of nodes based on the sub-graphs. To ensure the learned representations can effectively represent the characteristics of nodes, SEGEN need to ensure the sampled sub-graphs share similar properties as the original large-sized input graph. As shown in Figure 12, five different types of graph sampling strategies (indicated in five different colors) are adopted, and each strategy will lead to a group of small-sized sub-graphs, which can capture both the local and global structures of the original graph. According to [9], the sampled sub-graphs based on different sampling strategies, e.g., BFS, DFS, BNS, BES and HS, can be represented as $\mathcal{G}^{\text{BFS}}$, $\mathcal{G}^{\text{DFS}}$, $\mathcal{G}^{\text{HS}}$, $\mathcal{G}^{\text{NS}}$ or $\mathcal{G}^{\text{ES}}$, respectively.

Instead of fitting each unit model with all the sub-graphs in the pool $\mathcal{G}$, in the unit model, a set of sub-graph training batches $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m$ will be sampled for each unit model respectively in the learning process, where $|\mathcal{T}_i| = b, \forall i \in \{1, 2, \cdots, m\}$ are of the pre-defined batch size b. These batches may share common sub-graphs as well, i.e., $\mathcal{T}_i \cap \mathcal{T}_j$ may not necessary be $\emptyset$. In the model, the unit models learning process for each generation involves two steps: (1) generating the batches $\mathcal{T}_i$ from the pool set $\mathcal{G}$ for each unit model $M_i^1 \in \mathcal{M}^1$, and (2) learning the variables of the unit model M_i^1 based on sub-graphs in batch $\mathcal{T}_i$. Considering that the unit models have a much smaller number of hidden layers, the learning time cost of each unit model will be much less than the deeper models on larger-sized graphs.

In the following parts, we will first introduce the learning process of the model, which accepts each sub-graph pool as the input and learns the representation feature vectors of nodes as the output. We can use $\mathcal{G}$ to represent the sampled pool set, which can be $\mathcal{G}^{\text{BFS}}$, $\mathcal{G}^{\text{DFS}}$, $\mathcal{G}^{\text{HS}}$, $\mathcal{G}^{\text{NS}}$ or $\mathcal{G}^{\text{ES}}$ respectively. The learned results can be further fused together with a hierarchical ensemble process to be introduced at the last subsection.

3.6.2 Genetic Evolutionary Learning of seGEN

The training process of seGEN involves several key steps, including *unit model evaluation and selection*, *crossover* and *mutation*, which will be introduced as follows:

- **Evaluation and Selection**: The unit models in the generation set $\mathcal{M}^1$ can have different performance, due to (1) different initial variable values, and (2) different training batches in the learning process. In framework seGEN, instead of applying "deep" models with multiple hidden layers, it proposes to "deepen" the models in another way: "evolve the unit model into 'deeper' generations". For each unit model $M_k^1 \in \mathcal{M}^1$, based on the sub-graphs in a validation set $\mathcal{V}$, seGEN measures the introduced loss of the model as

  $$\ell(M_k^1; \mathcal{V}) = \sum_{g \in \mathcal{V}} \sum_{v_i, v_j \in \mathcal{V}_g, v_i \neq v_j} s_{i,j} \left\| \mathbf{z}_{k,i}^1 - \mathbf{z}_{k,j}^1 \right\|_2^2,$$

  where $\mathbf{z}_{k,i}^1$ and $\mathbf{z}_{k,j}^1$ denote the learned latent representation feature vectors of nodes v_i, v_j in the sampled sub-graph g. Term $s_{i,j}$ has value $+1$ if v_i and v_j are connected in subgraph g, otherwise $s_{i,j}$ will be assigned with value 0 instead.

  The probability for each unit model to be picked as the parent model for the *crossover* and *mutation* operations can be represented as

  $$p(M_k^1) = \frac{\exp^{-\ell(M_k^1; \mathcal{V})}}{\sum_{M_i^1 \in \mathcal{M}^1} \exp^{-\ell(M_i^1; \mathcal{V})}}.$$

  In the real-world applications, a normalization of the loss terms among these unit models is necessary. For the unit model introducing a smaller loss, it will have a larger chance to be selected as the parent unit model. Considering that the *crossover* is usually done based a pair of parent models, the pairs of parent models selected from set $\mathcal{M}^1$ can be denoted as $\mathcal{P}^1 = \{(M_i^1, M_j^1)_k\}_{k \in \{1,2,\cdots,m\}}$, based on which seGEN will be able to generate the next generation of unit models, i.e., $\mathcal{M}^2$.

- **Crossover**: For the k_{th} pair of parent unit model $(M_i^1, M_j^1)_k \in \mathcal{P}^1$, their genes can be denoted as their variables θ_i^1, θ_j^1 respectively (since the differences among the unit models mainly lie in their variables), which are actually their chromosomes for crossover and mutation.

  seGEN proposes to adopt the *uniform crossover* to get the chromosomes (i.e., the variables) of their child model. Considering that the parent models M_i^1 and M_j^1 can actually achieve different performance on the validation set $\mathcal{V}$, in the crossover, the unit model achieving better performance should have a larger chance to pass its chromosomes to the child model.

Formally, the chromosome inheritance probability for parent model M_i^1 can be represented as

$$p(M_i^1) = \frac{\exp^{-\ell(M_i^1;\mathcal{V})}}{\exp^{-\ell(M_i^1;\mathcal{V})} + \exp^{-\ell(M_j^1;\mathcal{V})}}$$

Meanwhile, the chromosome inheritance probability for model M_j^1 can be denoted as $p(M_j^1) = 1 - p(M_i^1)$.

In the uniform crossover method, based on parent model pair $(M_i^1, M_j^1)_k \in \mathcal{P}^1$, the obtained child model chromosome vector can be denoted as $\theta_k^2 \in \mathbb{R}^{|\theta^1|}$ (the superscript denotes the 2_{nd} generation and $|\theta^1|$ denotes the variable length), which is generated from the chromosome vectors θ_i^1 and θ_j^1 of the parent models. Meanwhile, the crossover choice at each position of the chromosomes vector can be represented as a vector $\mathbf{c} \in \{i, j\}^{|\theta^1|}$. The entries in vector $\mathbf{c}$ are randomly selected from values in $\{i, j\}$ with a probability $p(M_i^1)$ to pick value i and a probability $p(M_j^1)$ to pick value j respectively. The l_{th} entry of vector θ_k^2 before mutation can be represented as

$$\hat{\theta}_k^2(l) = \mathbb{1}\left(c(l) = i\right) \cdot \theta_i^1(l) + \mathbb{1}\left(c(l) = j\right) \cdot \theta_j^1(l),$$

where indicator function $\mathbb{1}(\cdot)$ returns value 1 if the condition is True; otherwise, it returns value 0.

- **Mutation**: The variables in the chromosome vector $\hat{\theta}_k^2(l) \in \mathbb{R}^{|\theta^1|}$ are all real values, and some of them can be altered, which is also called *mutation* in traditional genetic algorithm. Mutation happens rarely, and the chromosome mutation probability is γ in the model. Formally, the mutation indicator vector can be denoted as $\mathbf{m} \in \{0, 1\}^d$, and the l_{th} entry of vector θ_k^2 after mutation can be represented as

$$\theta_k^2(l) = \mathbb{1}\left(m(l) = 0\right) \cdot \hat{\theta}_k^2(l) + \mathbb{1}\left(c(l) = 1\right) \cdot rand(0, 1),$$

where $rand(0, 1)$ denotes a random value selected from range $[0, 1]$. Formally, the chromosome vector θ_k^2 defines a new unit model with knowledge inherited form the parent models, which can be denoted as M_k^2. Based on the parent model set $\mathcal{P}^1$, all these newly generated models can be represented as $\mathcal{M}^2 = \{M_k^2\}_{(M_i^1, M_j^1)_k \in \mathcal{P}^1}$, which will form the 2_{nd} generation of unit models.

3.6.3 Result Ensemble

Based on the models introduced in the previous subsection, seGEN adopts a hierarchical result ensemble method, which involves two steps: (1) *local ensemble* of results for the sub-graphs on each sampling strategies, and (2) *global ensemble* of results obtained across different sampling strategies.

- **Local Ensemble**: Based on the sub-graph pool $\mathcal{G}$ obtained via the sampling strategies introduced before, we have learned the K_{th} generation of the unit model $\mathcal{M}^K$ (or $\mathcal{M}$ for simplicity), which contains m unit models. Formally, given a sub-graph $g \in \mathcal{G}$ with node set $\mathcal{V}_g$, by applying unit model $M_j \in \mathcal{M}$ to g, the learned representation for node

$v_q \in \mathcal{V}_g$ can be denoted as vector $\mathbf{z}_{j,q}$, where q denotes the unique node index in the original complete graph G before sampling. For the nodes $v_p \notin \mathcal{V}_g$, its representation vector will be $\mathbf{z}_{j,p} = \mathbf{null}$, which denotes a dummy vector of length d. Formally, the learned representation feature vector for node v_q can be represented as

$$\mathbf{z}_q = \bigsqcup_{g \in \mathcal{G}, M_j \in \mathcal{M},} \mathbf{z}_{j,q}, \tag{54}$$

where operator $\sqcup$ denotes the concatenation operation of feature vectors. Considering that in the graph sampling step, not all nodes will be selected in sub-graphs. For the nodes $v_p \notin \mathcal{V}_g, \forall g \in \mathcal{G}$, SEGEN introduces a local propagation approach to compute its representations based on its neighbors instead.

Global Ensemble: Generally, these different graph sampling strategies can capture different local/global structures of the graph, which will all be useful for the node representation learning. In the global result ensemble step, SEGEN proposes to group these features together as the output. Formally, for node v_q in the original network, its fused representations can be denoted as

$$\bar{\mathbf{z}}_q = \sum_{i \in \{\text{BFS},\text{DFS},\text{HS},\text{NS},\text{ES}\}} w^i \cdot \mathbf{z}_q^i, \tag{55}$$

where $\mathbf{z}_q^{\text{BFS}}$, $\mathbf{z}_q^{\text{DFS}}$, $\mathbf{z}_q^{\text{HS}}$, $\mathbf{z}_q^{\text{NS}}$ and $\mathbf{z}_q^{\text{ES}}$ are the vectors of v_q obtained from the above local ensemble based on different graph sampling strategies. In [9], SEGEN will simply assign them with equal weights, i.e., $\bar{\mathbf{z}}_q$ is an average of $\mathbf{z}_q^{\text{BFS}}$, $\mathbf{z}_q^{\text{DFS}}$, $\mathbf{z}_q^{\text{HS}}$, $\mathbf{z}_q^{\text{NS}}$ and $\mathbf{z}_q^{\text{ES}}$.

4. Challenges and Opportunities

We have also observed many challenges with graph neural network studies, which provide plenty of opportunities for researchers interested in this topic:

4.1 Over-Smoothing Problem

The existing graph neural networks on giant network representation learning problem suffer from the *over-smoothing problem* a lot. For instance, if a GCN is deep with many convolutional layers, the output features may be over-smoothed and vertices from different clusters may become indistinguishable, which will render the GCN model fail to work. We have also observed some approaches proposed to resolve such a problem. In [1, 8], both the DIFNN and GNL models introduce a set of gates (i.e., the GDU neuron unit) to ensure the nodes can capture the raw inputs, neighbors' influences and the temporal dynamic states in the learning process, which can resolve the over-smoothing problem effectively.

4.2 Optimization Efficiency

The time complexity of the graph neural network learning, including those for small graphs and giant networks, can be very high. For the small graph oriented graph neural networks, e.g., ISONN, the major time is spent on enumerating the permutation matrices for the isomorphic feature calculation. Meanwhile, for the giant network oriented graph neural networks, e.g., GCN and GAT, most of the time is spent on the backpropagation along the graph edges for the nodes, which grows almost quadratically as the network size increases and exponentially as the model goes deeper. New optimization algorithms that can address the high learning cost will be necessary and desired.

4.3 The Gap Between Small Graphs and Giant Networks

By this context so far, we haven't witnessed any graph neural networks that can learn effective representations for both the small graphs and the giant networks simultaneously without any architecture modifications. Proposing a new unified graph neural network model that can work for different types of networks will be desired.

4.4 Graph Neural Network for Dynamic Networks

Most of the network data in the real-world are not static, which keep changing with time. We have observed the GNL model [1] as the first work focusing on the dynamic regression scenario. Such kinds of model can be important, and it may also serve as the tool for analyzing and understanding the brain activities, which is a dynamic network with both structure and states changing all the time.

4.5 Graph Neural Network for Complex Networks

Most of the graph neural networks proposed so far mainly focus on the homogeneous network, which over-simplify the learning setting, since most of the network data in the real world are heterogeneous instead. Generally, in heterogeneous networks, different types of nodes and links can convey different kinds of physical meanings. New models that can effective incorporate such heterogeneous information in the learning process can be desired for concrete real-world applications of graph neural networks.

5. Summary

In this paper, we have introduced the latest graph neural networks proposed for resolving the small graph and giant network oriented research problems. The small graph oriented graph neural network models introduced in this paper include IsoNN, SDBN and LF&ER; whereas, the giant network oriented graph neural network models introduced in this paper include GCN, GAT, DifNN, GNL, GraphSage and seGEN. In addition, we have also introduced several challenges and opportunities on graph neural network studies. This paper will be updated shortly as we observe the new development on this topic in the near future.

References

[1] Yixin Chen, Lin Meng, and Jiawei Zhang. Graph neural lasso for dynamic network regression. *CoRR*, abs/1907.11114, 2019.

[2] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *CoRR*, abs/1706.02216, 2017.

[3] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[4] Lin Meng and Jiawei Zhang. Isonn: Isomorphic neural network for graph representation learning and classification. *CoRR*, abs/1907.09495, 2019.

[5] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[6] Pengyang Wang, Jiawei Zhang, Guannan Liu, Yanjie Fu, and Charu Aggarwal. *Ensemble-Spotting: Ranking Urban Vibrancy via POI Embedding with Multi-view Spatial Graphs*, pages 351–359.

[7] Shen Wang, Lifang He, Bokai Cao, Chun-Ta Lu, Philip S. Yu, and Ann B. Ragin. Structural deep brain network mining. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 475–484, New York, NY, USA, 2017. ACM.

[8] Jiawei Zhang, Limeng Cui, Yanjie Fu, and Fisher B. Gouza. Fake news detection with deep diffusive network model. *CoRR*, abs/1805.08751, 2018.

[9] Jiawei Zhang, Limeng Cui, and Fisher B. Gouza. SEGEN: sample-ensemble genetic evolutional network model. *CoRR*, abs/1803.08631, 2018.