Some Mathematical Perspectives of Graph Neural Networks

by

Duy Nguyen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Applied Mathematics

Waterloo, Ontario, Canada, 2022
© Duy Nguyen 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Many real-world entities can be modelled as graphs, such as molecular structures, social networks, or images. Despite coming with such a great expressive power, the complex structure of graphs poses significant challenges to traditional deep learning methods, which have been extremely successful in many machine learning tasks on other input data structures, such as texts and images data. Recently, there have been many attempts in developing neural network architectures on graphical data, namely graph neural networks (GNNs). In this thesis, we first introduce some mathematical notations for graphs and different aspects of training a feedforward neural network. We then discuss several notable GNN architectures including Graph Convolutional Neural Networks, Graph Attention Networks, GraphSAGE, and PinSAGE. Some special aspects of GNN training are also presented. Finally, we investigate a neighborhood sampling approach on PinSAGE to a product-user recommendation problem.

Acknowledgements

I would like to thank all the little people who made this thesis possible.

Table of Contents

List of Figures viii

List of Tables x

1 Introduction 1

2 Background 3

2.1 Graphs 3

2.2 Machine Learning and Neural Networks Basics 5

2.2.1 Tasks 6

2.2.2 Performance Measure 7

2.2.3 Loss Function 8

2.2.4 Parameter Updates 9

2.2.5 Neural Networks 11

2.2.6 Overfitting and Regularization 12

2.3 Fourier Transform and Convolution Theorem 13

3 Graph Neural Networks Architecture 15

3.1 Graph Convolutional Network (GCN) 15

3.1.1 Spectral Motivation 16

3.1.2 Spatial Motivation 18

3.1.3 Efficient Implementation 20
3.2 GraphSAGE 20
3.3 Graph Attention Network (GAT) 22
3.4 PinSAGE 24

4 GNN Training **27**
4.1 Backpropagation 27
4.1.1 Backpropagation on GCNs 27
4.1.2 Skip Connections in GNNs 29
4.2 Loss Functions 30
4.3 Special Training Methods on Graphs 32
4.3.1 Mini-Batching 32
4.3.2 Sampling 32
4.3.3 Graph Pooling 34
4.3.4 Augmentation 34
4.3.5 Regularization on Graphs 35

5 Numerical Experiments **37**
5.1 Dataset and Task 37
5.1.1 Data Format 37
5.1.2 Task 39
5.2 Environment 39
5.2.1 Libraries 39
5.2.2 Experiment Environment 39
5.3 General Approach 40
5.3.1 Dataset Split 40
5.3.2 Preprocessing 41
5.3.3 Constructing Samplers 41

5.3.4 Constructing a Training Batch 41
5.3.5 Scoring and Loss Functions 42
5.3.6 Model Architecture 42
5.4 Evaluation 43
5.4.1 Hit@K 43
5.4.2 Evaluation Settings 43
5.5 Results and Extensions 43
5.5.1 Baseline Model 43
5.5.2 Adaptive Restart Probability 44
5.5.3 Variable Number of Random Walks 44
5.6 Observations 45

6 Conclusions **51**
6.1 Summary 51
6.2 Future Work 51

References **53**

List of Figures

2.1 A fully-connected neural network with two hidden layers. 11

3.1 GCN as a spatial architecture - information is passed from a node to its direct neighbors. Information of node 5 at ℓ^{th} layer is passed to nodes 1 and 4 at $(\ell+1)^{th}$ layer. The hidden representation of node 2 at $(\ell+1)^{th}$ layer is calculated based on the hidden representation of nodes 1 and 3 at ℓ^{th} layer. 19

3.2 Visualization of GraphSAGE. **Left**: Sampled neighborhood of node 0, $\mathcal{N}_0 = \{1, 3, 4, 5\}$. **Right**: Node 0 aggregates information from its neighborhood. 20

3.3 GAT: From hidden representation to attention weights (See Equations (3.22) - (3.24)) 23

3.4 An illustration of multihead attention (with K = 3 heads) by node 1 on its neighborhood. Different colors represent different heads. The aggregated features from each head are concatenated or averaged to obtain h'_1, which is the representation of node 1 at the next layer. The figure is taken from [20]. 23

3.5 Visualization of PinSAGE's importance-based neighborhood sampling. **Left**: Original graph **Center**: Simulation of $N = 5$ random walks from node 0, each with length $L = 2$, top $T = 3$ is chosen. **Right**: Node 0 aggregates information from its sampled neighborhood. 25

4.1 Effect of adding skip connections in graph neural networks: adding a skip connection (blue arrow) only helps mitigate the vanishing gradient risk from the node itself. The risk of vanishing gradient from message passing through its neighbors still remains. Figure is taken from [15] 29

4.2 DiffPool [23] for graph classifcation problems. At each hierarchical layer, a GNN is run to obtain embeddings of nodes. The learn learned embeddings are used to cluster nodes together. Then another GNN layer is run on this coarsened graph. This whole process is repeated for L layers and the final output representation is used to classify the graph. 34

5.1 Numerical result for Experiment 1 (Original PinSAGE), K = 5 46

5.2 Numerical result for Experiment 2 (Original PinSAGE), K = 50 46

5.3 Numerical result for Experiment 3 (Original PinSAGE), K = 200 47

5.4 Experiment 4: Original PinSAGE vs Adaptive Restart Probability method 47

5.5 Experiment 5: Original PinSAGE vs Variable Number of Random Walks method 48

List of Tables

1 Graph Neural Networks Notation xi

5.1 User Features 38

5.2 Movie Features 38

5.3 Rating Features 38

5.4 List of Python libraries used and their versions 39

5.5 Machine configuration 40

5.6 Hyperparameters for the original PinSAGE (baseline) 48

5.7 Hyperparameters for the Adaptive Restart Probability (ARP) method . . . 49

5.8 Hyperparameters for the Variable Random Walk Lengths (VRWL) method 49

5.9 Random seeds set for libraries/packages that involve randomness 50

Table 1: Graph Neural Networks Notation

Notation	**Description**
$\mathbf{I}_N \in \mathbb{R}^{N\times N}$	Identity matrix of dimension N
$G = (V, E)$	Graph G with vertex set V and edge set E
N	Number of nodes in the graph
u, v	Vertices (nodes) in the graph
$\alpha_{uv} \in \mathbb{R}$	Importance of node v to node u
L	Number of layers in the graph neural network
$[L]$	The set $\{1, .., L\}$
$n^{(\ell)} \in \mathbb{R}$	Number of hidden units at layer ℓ
$\mathcal{N}(v)$	Set of neighboring nodes of v
$\mathcal{N}_s(v)$	Set of sampled neighboring nodes of v
$\mathbf{L} \in \mathbb{R}^{N\times N}$	Laplacian matrix of graph G
$\mathbf{A} \in \mathbb{R}^{N\times N}$	Adjacency matrix of graph G
$\widehat{\mathbf{A}} \in \mathbb{R}^{N\times N}$	Adjacency matrix with self-loop added
$\breve{\mathbf{A}} \in \mathbb{R}^{N\times N}$	Normalized adjacency matrix with self-loop added
$\mathbf{D} \in \mathbb{R}^{N\times N}$	Node degree matrix associated with $\mathbf{A}$
$\widehat{\mathbf{D}} \in \mathbb{R}^{N\times N}$.	Node degree matrix associated with $\widehat{\mathbf{A}}$
$\mathbf{X} \in \mathbb{R}^{N\times d}$	Input feature matrix
$\mathbf{Z} \in \mathbb{R}^{N\times n^{(L)}}$	Prediction head matrix
$\mathbf{H}^{(\ell)} \in \mathbb{R}^{N\times n^{(\ell)}}$	Hidden state matrix (output) at layer ℓ
$\sigma(\cdot)$	A nonlinear activation function
$\mathbf{W}^{(\ell)} \in \mathbb{R}^{n^{(\ell+1)}\times n^{(\ell)}}$	Trainable weight matrix at layer ℓ
$\mathbf{h}_v^{(\ell)} \in \mathbb{R}^{n^{(\ell)}}$	Hidden state vector of node v at layer ℓ, can also be referred as embedding of node v at layer ℓ
$\mathbf{a}_\ell \in \mathbb{R}^{2n^{(\ell+1)}}$	Attention mechanism at layer ℓ
$\odot$	Element-wise multiplication
$\|\|$	Concatenation operator
$*$	Convolution operator

Chapter 1

Introduction

Many real-world entities can be represented as graphs, such as social networks, molecular structures, or images. For example, a molecule can be viewed as a graph of many atoms bonded together, and their properties needed to be predicted for drug discovery. In e-commerce, users and products can be represented as nodes in a graph whose edges model interaction among them, and the goal is to exploit the complex structure of these interactions to serve accurate recommendations to users. The first motivation of Graph Neural Networks (GNNs) comes from the nineties, but they have just been gaining massive attraction recently, following the success of convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Many operations on graphs are generalized from their successful counterparts in CNNs and RNNs, such as graph convolution and graph attention. Thanks to recent advancement in computing powers, many generalizations, definitions and extensions have been proposed and experimented, which leaves a spacious room of exploration and improvements for researchers and practitioners. However, partly because of that advancement, results in the field of GNNs are almost always of a computational nature, somewhat lacking rigorous mathematical foundation. There is also discrepancy in the mathematical representations among GNN architectures, results in confusion from readers.

In this thesis, we seek to present a systematic way to write down popular GNN architectures, and the mathematical proof/foundation that is lacking in some original works. Chapter 2 introduces useful notations oh graphs, machine learning tasks in general, and different aspects of training a neural network. This chapter is followed by an overview of several most notable GNN architectures, including Graph Convolutional Network (GCN), GraphSAGE, Graph Attention Network (GAT), and PinSAGE. Motivation, strengths, weaknesses and use cases of each architecture are also discussed. Some specific aspects of training a GNN

are presented and discussed in Chapter 4. Finally, we investigate a neighborhood sampling approach based on PinSAGE to a product recommendation problem, report our findings, discuss the numerical results, and propose some improvements.

Chapter 2

Background

This section introduces some useful definitions and properties of graphs [9] and neural networks that will be used for subsequent chapters. Firstly, we define graphs and introduce several graph terminologies, including adjacency matrices, node degree matrices, and Laplacian matrices. Then we provide an overview of basic principles of machine learning and neural networks, including tasks, performance measure, and training aspects. Finally, we recall the Fourier transform and convolution theorem, which serve as inspirations for the graph convolutional operator.

2.1 Graphs

Definition 1 (Graph)**.** *A **graph** $G = (V, E)$ is defined as a collection of two sets V and E, where V is a finite set of nodes and $E \subseteq V \times V$ is the set of edges. Here we say there is an edge between node $u \in V$ and node $v \in V$ if $(u, v) \in E$.*

*The graph G is called a **simple graph** if there is at most one edge between any pair of nodes.*

*The graph G is called an **undirected graph** if the edges are undirected, i.e. $(u, v) \in E$ if and only if $(v, u) \in E$.*

*Let $v \in V$ be a node of a graph $G = (V, E)$. We define the **set of neighboring nodes** of v, $\mathcal{N}(v)$, as follows:*

$$\mathcal{N}(v) = \{u \in V | (u, v) \in E\}.$$

Definition 2 (Adjacency matrix)**.** *Let $G = (V, E)$ be a simple undirected graph, where $V = \{v_1, \ldots, v_N\}$. The **adjacency matrix** $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{N \times N}$ of the graph G is defined to be the matrix whose entries are:*

$$a_{ij} = \begin{cases} 1, & \textit{if } (v_i, v_j) \in E \\ 0, & \textit{otherwise,} \end{cases}$$

for $i, j \in [N]$.

Definition 3 (Node degree matrix)**.** *Given a simple undirected graph $G = (V, E)$, we define its **node degree matrix** $\mathbf{D} = (d_{ij}) \in \mathbb{R}^{N \times N}$ as a diagonal matrix whose diagonal entries are:*

$$d_{ii} = \sum_{j=1}^{N} a_{ij} \geqslant 0, \quad i \in [N]$$

where (a_{ij}) is the adjacency matrix of G.

Definition 4 (Adjacency matrix with self-connections)**.** *Let $G = (V, E)$ be a simple undirected graph, where $V = \{v_1, \ldots, v_N\}$. We define its **adjacency matrix with self-connections** $\widehat{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N \in \mathbb{R}^{N \times N}$, where $\mathbf{A}$ is the adjacency matrix of graph G. Explicitly, the entry $\widehat{a}_{ij}$ of $\widehat{\mathbf{A}}$, where $i, j \in [N]$, is*

$$\widehat{a}_{ij} = \begin{cases} 1, & \textit{if } (v_i, v_j) \in E \textit{ or } i = j, \\ 0, & \textit{otherwise.} \end{cases}$$

Definition 5 (Node degree matrix with self-connections)**.** *Let $G = (V, E)$ be a simple undirected graph, where $V = \{v_1, \ldots, v_N\}$. We define its **node degree matrix with self-connections** $\widehat{\mathbf{D}} = (\widehat{d}_{ij}) \in \mathbb{R}^{N \times N}$ as a diagonal matrix whose diagonal entries are:*

$$\widehat{d}_{ii} = \sum_{j=1}^{N} \widehat{a}_{ij} \geqslant 0, \quad i \in [N],$$

where $\widehat{\mathbf{A}} = (\widehat{a}_{ij})$ is the adjacency matrix with self-connections of the graph G.

Definition 6 (Normalized adjacency matrix with self-connections)**.** *Let $G = (V, E)$ be a simple undirected graph, where $V = \{v_1, \ldots, v_N\}$. We define its **normalized adjacency matrix** $\breve{\mathbf{A}} = \widehat{\mathbf{D}}^{-\frac{1}{2}} \widehat{\mathbf{A}} \widehat{\mathbf{D}}^{-\frac{1}{2}} \in \mathbb{R}^{N \times N}$ with the convention that $0^{-\frac{1}{2}} = 0$. Here $\widehat{D}$ and $\widehat{A}$ are the node degree matrix with self-connections and the adjacency matrix with self-connections of the graph G, respectively.*

Definition 7 (Laplacian matrix)**.** *Given a simple undirected graph $G = (V, E)$, the Laplacian matrix* $\mathbf{L}$ *and the symmetrically normalized Laplacian matrix* $\mathbf{L}^{sym}$ *of graph G are defined by:*

$$\mathbf{L} := \mathbf{D} - \mathbf{A} \in \mathbb{R}^{N \times N},$$
$$\mathbf{L}^{sym} := \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}} = \mathbf{I}_N - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} \in \mathbb{R}^{N \times N},$$

where $\mathbf{D}, \mathbf{A}$ *are the node degree matrix and adjacency matrix of the graph G, respectively, with the convention that* $0^{-\frac{1}{2}} = 0$.

Notice that since the graph G is undirected, its adjacency matrix A is symmetric. Therefore, the Laplacian matrices $\mathbf{L}$ and $\mathbf{L}^{sym}$ are also symmetric. Moreover, since $\mathbf{L}$ is diagonally dominant, $\mathbf{L}$ is positive semidefinite. Since $\mathbf{L}^{sym} = \mathbf{D}^{-1/2}\mathbf{L}\mathbf{D}^{-1/2}$ and both $\mathbf{D}^{-\frac{1}{2}}$ and $\mathbf{L}$ are positive semidefinite, $\mathbf{L}^{sym}$ is also positive semidefinite. Using $\mathbf{L}^{sym}$ over $\mathbf{L}$ for messaging passing has several advantages. Firstly, it makes the influence of neighboring nodes with large degrees of a node u more equal to that of other neighboring nodes. Moreover, while $\mathbf{L}$ and $\mathbf{L}^{sym}$ share the same set of eigenvectors, $\mathbf{L}^{sym}$ has a bounded spectrum, which ensures several numerical stability properties [9]. More precisely, it can be proved that $0 = \lambda_1 \leqslant ... \leqslant \lambda_N \leqslant 2$ where $\{\lambda_i\}_{i=1}^{N}$ be the set of eigenvalues of $\mathbf{L}^{sym}$ [5].

Definition 8 (Laplacian matrix with self-connections)**.** *Given a simple undirected graph $G = (V, E)$, we define the Laplacian matrix with self-connections* $\widehat{\mathbf{L}}$ *and the symmetrically normalized Laplacian matrix with self-connection* $\widehat{\mathbf{L}}^{sym}$ *of graph G as follows:*

$$\widehat{\mathbf{L}} := \widehat{\mathbf{D}} - \widehat{\mathbf{A}} \in \mathbb{R}^{N \times N},$$
$$\widehat{\mathbf{L}}^{sym} := \widehat{\mathbf{D}}^{-1/2}\widehat{\mathbf{L}}\widehat{\mathbf{D}}^{-1/2} = \mathbf{I}_N - \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}} \in \mathbb{R}^{N \times N}$$

where $\widehat{\mathbf{D}}$ *and* $\widehat{\mathbf{A}}$ *are the node degree matrix with self-connections and adjacency matrix with self-connections, respectively.*

For the remaining of the thesis, we will mainly use the symmetrically normalized Laplacian matrices with self-connections $\widehat{\mathbf{L}}^{sym}$. The advantages of using matrices with self-connections will be discussed in Section 3.1.

2.2 Machine Learning and Neural Networks Basics

In this section, we will provide an overview of basic principles of machine learning, deep learning, and neural networks, based on [7].

2.2.1 Tasks

We can categorize machine learning tasks as **supervised**, **unsupervised**, or **semi- supervised** learning.

In **supervised learning**, we aim to learn an underlying function that maps the input (feature vectors) to the corresponding output (labels) from a given set of input-output pairs and make predictions for unseen examples. Two of the most common tasks for supervised learning algorithms are classification and regression. In **classification** tasks, the algorithm aims to predict which of C given categories that an input belongs to. To make such prediction, the algorithm learns a function $f : \mathbb{R}^n \rightarrow \{1, \ldots, C\}$, where n is the dimension of each input data. For example, the spam filtering problem is a classification task, where the algorithm may be given a dataset of content of the emails, name of the sender, etc. and is required to predict whether that email is spam or not. In **regression** tasks, the output can be numerical values. To make such prediction, the algorithm try to learn a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where n and m are the dimension of the input and output data, respectively. For example, the house price prediction problem is a regression task, where each input data contains the properties of a house (size, location, etc.) and the corresponding output data is the sold price of the house. The regression algorithm will learn a function (possibly nonlinear) that maps the properties of the house to its sold price and predict the prices of new houses.

In **unsupervised learning**, we try to find underlying patterns from a given dataset. Some unsupervised learning algorithms want to learn the probability distribution that generated the dataset, while some others aim to divide the dataset into clusters of similar inputs (**clustering**). One example is the StackOverflow questions clustering problem in which we would like to group similar questions into the same cluster to recommend them to users.

In **semi-supervised learning** tasks, we are given a dataset of features, where a subset of the dataset is labelled and the rest are unlabeled. For graph inputs, we are also given a graph that contains nodes and edges, and a semi-supervised task can be done in transductive setting or inductive setting. In **transductive setting**, the model is required to predict the labels of the given unlabeled nodes. One example is the citation network problem [11], where we are given a graph of research papers as nodes and citation relations as edges, along with labels (topics) for some research papers. The goal is to predict the topics of the remaining unlabeled papers. In **inductive setting**, the model is required to predict the labels for new nodes from the same distribution [24]. For example, we may train a graph neural network on a subgraph of the citation network then test its performance on another subgraph [9].

2.2.2 Performance Measure

A performance measure is a metric that can be used to evaluate the abilities of a machine learning algorithm. The list of choices for performance measures usually depends on the given task [7].

One of the most common machine learning tasks on graphs is classification. For **classification** tasks, there are many metrics that can be used to measure an algorithm's performance. One of the most common metric is **accuracy**. Accuracy is the proportion of correct predictions, over the total number of predictions. Equivalently, we can also use **error rate**, which is the proportion of incorrect predictions, over the total number of predictions. We can also use **precision**, **recall (sensitivity)**, **specificity**, where the definitions are given below.

The **precision** of a model is defined as follows:

$$Precision = \frac{TP}{TP + FP}.$$

The **recall** of a model is defined as follows:

$$Recall = \frac{TP}{TP + FN}.$$

The **specificity** of a model is defined as follows:

$$Specificity = \frac{TN}{TN + FP}.$$

Suppose we have a recommendation model where it predict whether a user will buy a given product or not. A **true positive** (TP) case is a case where we predict yes, and the user actually buys it. A **true negative** (TN) case is a case where we predict no, and the user actually does not buy it, as expected. A **false positive** (FP) case is a case where the user does not buy the product although we predict they do. A **false negative** (FN) case is a case where the user buys a product that we predict they do not.

Since there is usually a trade-off among precision, recall and specificity, we may want to choose to optimize different metrics in different scenarios. We want precision to be high when we cannot tolerate a false positive case. We want to optimize for recall if we do not want to miss a positive case. For instance, it is dangerous to miss a virus-infected person in case of a highly contagious disease. We want to optimize for specificity if we want to

avoid false alarms. For example, we want a cancer diagnosis to have high specificity as it costs a lot of time, money, and even patient's wellness to undergo cancer treatments.

On the other hand, it can be argued that we should only consider the performance of the machine learning algorithms on data that it has not seen before. For that reason, we can divide the given dataset into **training set** and **test set**. Training set is used by the algorithm to learn, while **test set** is reserved for evaluating the performance of the machine learning algorithm. In practice, the size of training dataset is usually bigger than the size of test dataset.

2.2.3 Loss Function

Most machine learning algorithms include solving optimization problems [7]. For example, in supervised learning settings, we usually want to minimize the loss function that measures the difference between the truth outputs and the outputs of the learning algorithm. Intuitively, the loss will be high if the algorithm makes poor predictions, and it will be low if the algorithm is doing well [13].

For example, in a classification task, we use a neural network which maps the feature vector $\mathbf{x}_i \in \mathbb{R}^d$ to the output $\mathbf{z}_i = [z_1, .., z_C]^T \in \mathbb{R}^C$ as the probability of the ith example falling onto C different classes, where $i \in [N]$. Denote $\mathbf{y}_i \in \mathbb{R}^C$ as the one-hot vector indicating the ground-truth class of the ith example. For instance, if the true class of the ith example is $k \in [C]$ then $\mathbf{y}_i = [y_{i1}, \ldots, y_{iC}]^T \in \mathbb{R}^C$ where:

$$\mathbf{y}_{ij} = \begin{cases} 1, & \text{if } j = k \\ 0, & \text{otherwise.} \end{cases}$$

For such classification tasks, minimizing the cross-entropy loss, which is equivalent to maximize the log-likelihood estimation of the data given the model, is one of the most popular choices. The cross-entropy loss is calculated as follows:

$$\mathcal{L} = -\sum_{i=1}^{N} \sum_{j=1}^{C} \mathbf{y}_{ij} \ln \mathbf{z}_{ij}. \tag{2.1}$$

For regression tasks, **Root Mean Square Error** (RMSE) is one of the most commonly used loss. It measures how far the model's predictions are from the actual values. Let $y_i \in \mathbb{R}$ be the actual value of the ith example, $\hat{y}_i$ is the predicted value, then the RMSE loss is

given by:

$$\mathcal{L} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}. \tag{2.2}$$

2.2.4 Parameter Updates

In this section, we discuss popular parameter updating methods in neural networks using first-order optimization algorithms. The list includes vanila gradient descent, gradient descent with momentum, Adagrad, RMSprop, and Adam. First, we introduce some useful notations.

Let $\mathbf{x} = (x_1, \ldots, x_N)^T \in \mathbb{R}^N$ and $\mathcal{L} : \mathbb{R}^N \to \mathbb{R}$. Then, the gradient of $\mathcal{L}$ with respect to $\mathbf{x}$ is:

$$\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}) = \left[\frac{\partial \mathcal{L}}{\partial x_1}(\mathbf{x}), \ldots, \frac{\partial \mathcal{L}}{\partial x_N}(\mathbf{x})\right]^T \in \mathbb{R}^N.$$

We define $\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}) \odot \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x})$ as an element-wise multiplication operation:

$$\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}) \odot \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}) = \left[\left(\frac{\partial \mathcal{L}}{\partial x_1}(\mathbf{x})\right)^2, \ldots, \left(\frac{\partial \mathcal{L}}{\partial x_N}(\mathbf{x})\right)^2\right]^T \in \mathbb{R}^N.$$

Denote t (where $t = 1, 2, \ldots$) the iteration count and $\mathbf{x}^{(t)} \in \mathbb{R}^N$ the vector we need to update. Then the vector $\mathbf{x}^{(t)}$ can be updated as follows.

- **Vanilla gradient descent update:**

  $$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} - \lambda \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}), \tag{2.3}$$

  where $\lambda > 0$ is the learning rate.

- **Gradient descent with momentum:** A momentum term $\mathbf{v} \in \mathbb{R}^N$ is added and initialized at zero, meanwhile $\beta \in [0, 1]$ is the decay factor that determines the contribution of the current gradient and previous gradients (momentum) to the weight change. Then the updates are given by:

  $$\mathbf{v}^{(t)} = \beta \mathbf{v}^{(t-1)} + (1-\beta)\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}), \tag{2.4}$$
  $$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} + \mathbf{v}^{(t)}. \tag{2.5}$$

- **Adagrad:** The main idea of this optimization method is to have per-parameter learning rates. That is, the component that receive high gradients will have their learning rate reduced over time. Here, $\mathbf{c}^{(t)} \in \mathbb{R}^N$ is the accumulated magnitude of gradients over time, and $\epsilon > 0$ is used to avoid division by zero. For each backprogagation step, we update

$$\mathbf{c}^{(t)} = \mathbf{c}^{(t-1)} + \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \odot \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \tag{2.6}$$

$$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} - \lambda \frac{\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \odot \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)})}{\sqrt{\mathbf{c}^{(t)}} + \epsilon}, \tag{2.7}$$

where the division here is element-wise division. Explicitly, we have

$$c_i^{(t)} = c_i^{(t-1)} + \nabla_{\mathbf{x}}\mathcal{L}(x_i^{(t-1)}) \odot \nabla_{\mathbf{x}}\mathcal{L}(x_i^{(t-1)}) \tag{2.8}$$

$$x_i^{(t)} = x_i^{(t-1)} - \lambda \frac{\nabla_{\mathbf{x}}\mathcal{L}(x_i^{(t-1)})}{\sqrt{c_i^{(t)}} + \epsilon}, \tag{2.9}$$

where $i \in [N], \mathbf{c}^{(t)} = \left(c_1^{(t)}, \ldots, c_N^{(t)}\right)^T$, and $\mathbf{x}^{(t)} = \left(x_1^{(t)}, \ldots, x_N^{(t)}\right)^T$.

- **RMSprop:** Similar to Adagrad, RMSProp also has per-parameter learning rates. The difference is that RMSprop uses a moving average of squared gradients instead of the instantaneous squared gradient. The updates are given by

$$\mathbf{c}^{(t)} = \beta\mathbf{c}^{(t-1)} + (1-\beta)\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \odot \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \tag{2.10}$$

$$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} - \lambda \frac{\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)})}{\sqrt{\mathbf{c}^{(t)}} + \epsilon}, \tag{2.11}$$

where division and square root are both performed element-wise, $\beta \in [0, 1]$ is the decay factor, and λ is the learning rate. In practice, β is usually set close to 1.

- **Adam:** Similar to RMSProp, Adam's method also uses a moving average for squared gradients. The main difference is that both the moving averages of the gradients $\mathbf{m}^{(t)} \in \mathbb{R}^N$ and squared gradients $\mathbf{v}^{(t)} \in \mathbb{R}^N$ are further smoothed out over time by a factor of $\beta_1 \in [0, 1]$ and $\beta_2 \in [0, 1]$. Usually, β_1 is set to 0.9 and β_2 is set to 0.99.

Both $\mathbf{v}$ and $\mathbf{m}$ are initialized at zero. The updates are given by

$$\mathbf{m}^{(t)} = \beta_1 \mathbf{m}^{(t-1)} + (1-\beta_1)\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \tag{2.12}$$

$$\widehat{\mathbf{m}}^{(t)} = \frac{\mathbf{m}^{(t)}}{1-\beta_1^t} \tag{2.13}$$

$$\mathbf{v}^{(t)} = \beta_2 \mathbf{v}^{(t-1)} + (1-\beta_2)\nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \odot \nabla_{\mathbf{x}}\mathcal{L}(\mathbf{x}^{(t-1)}) \tag{2.14}$$

$$\widehat{\mathbf{v}}^{(t)} = \frac{\mathbf{v}^{(t)}}{1-\beta_2^t} \tag{2.15}$$

$$\mathbf{x}^{(t)} = \mathbf{x}^{(t-1)} - \alpha \frac{\widehat{\mathbf{m}}^{(t)}}{\sqrt{\widehat{\mathbf{v}}^{(t)}} + \epsilon} \tag{2.16}$$

where the division and square-root are both element-wise division. In practice, Adam works really well, and currently it is the recommended algorithm for most problems.

2.2.5 Neural Networks

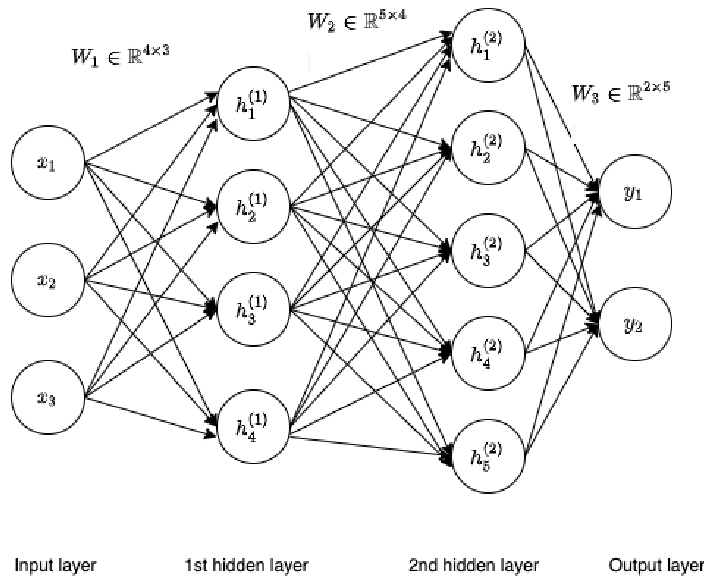


Figure 2.1: A fully-connected neural network with two hidden layers.

A neural network consists of neurons that are connected in an acyclic graph, where the outputs of some neurons can become inputs to other neurons. Usually, a neural network

is organized into distinct layers of neurons. The input data is fed into the first layer, and the output (usually the prediction we need to make from the input) is calculated based on output of the last layer. A neural network is typically initiated with random weights on the edges between nodes, these weights are then updated by the backpropagation algorithm repeatedly until the model performs well enough [14].

For example, let $\mathbf{x} \in \mathbb{R}^3$ be an input data and $\mathbf{y} \in \mathbb{R}^2$ be the corresponding output of a two-hidden layer neural network associated with Figure 2.1. The output of each layer is given by:

$$\begin{aligned} \mathbf{h}^{(1)} &= \sigma(\mathbf{W}_1 x + \mathbf{b}_1) \in \mathbb{R}^4 \\ \mathbf{h}^{(2)} &= \sigma(\mathbf{W}_2 \mathbf{h}^{(1)} + \mathbf{b}_2) \in \mathbb{R}^5 \\ \mathbf{y} &= \mathbf{W}_3 \mathbf{h}^{(2)} + \mathbf{b}_3 \in \mathbb{R}^2, \end{aligned} \tag{2.17}$$

where $\mathbf{W}_1 \in \mathbb{R}^{4\times 3}, \mathbf{W}_2 \in \mathbb{R}^{5\times 4}, \mathbf{W}_3 \in \mathbb{R}^{2\times 5}, \mathbf{b}_1 \in \mathbb{R}^4, \mathbf{b}_2 \in \mathbb{R}^5, \mathbf{b}^3 \in \mathbb{R}^2$, and σ is a nonlinear activation function.

In general, a feedforward L-hidden-layer neural network is a function of the form:

$$f(\mathbf{x};\boldsymbol{\theta}) = \mathbf{W}_L \sigma\Big(\mathbf{W}_{L-1}\sigma\big(\ldots\sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\ldots\big) + \mathbf{b}_{L-1}\Big) + \mathbf{b}_L, \tag{2.18}$$

where $\boldsymbol{\theta} = \{(\mathbf{W}_\ell, \mathbf{b}_\ell)\}_{\ell=1}^{L}, \mathbf{W}_\ell \in \mathbb{R}^{n^{(\ell)}\times n^{(\ell+1)}}, \mathbf{b}_\ell \in \mathbb{R}^{n^{(\ell)}}, n^{(l)} \in \mathbb{Z}^+$, and $\ell \in [L]$.

2.2.6 Overfitting and Regularization

According to [7], the performance of a machine learning algorithms is measured by how good their outputs for examples in the test set are. It can be decomposed into two factors: how big the training error is, and how big the gap between training error and test error is. **Underfitting** is the phenomenal when the model cannot perform well on training set. One of the way to combat underfitting is to increase the number of trainable parameters. On the other hand, **overfitting** is when there is a huge discrepancy between training error and test error. **Regularization** is a set of techniques that can be used to prevent overfitting.

For the remaining of this section, we discuss regularization methods in neural network settings. To make a neural network better fit a training set, people can increase the number of learnable parameters or expand the search range of them, thus make the model more capable of "memorizing" the training set. On the contrary side, to prevent overfitting, we can decrease the number of learnable parameters, or restrict the ranges of these parameters. The idea is inspired from the Occam's razor principle. This principle states that among different hypotheses that explain an observation equally well, we should choose the simplest

one. Translating to machine learning settings, the idea is to give more preference to simple or sparse set of parameters, and penalize overcomplex models. Here are several examples:

Suppose we are using cross-entropy loss as in (2.1). We want to regularize all the weight matrices $\mathbf{W}^{(\ell)}$ where $\ell \in [L]$. If we prefer the weight matrices to be sparse, we can use **L1 regularization**. Because of this property, L1 regularization can also be viewed as a feature selection mechanism [7]. The loss function with incorporated L1 regularization term is given by:

$$\mathcal{L} = -\sum_{u=1}^{N}\sum_{i=1}^{C}\mathbf{y}_{ui}\ln\mathbf{z}_{ui} + \sum_{\ell=1}^{L}\sum_{i}\sum_{j}|\mathbf{W}_{ij}^{(\ell)}|. \tag{2.19}$$

Another popular choice of regularization is to use the Frobenius norm on the weight matrices, which is also called the weight decay (**L2 regularization**):

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{j=1}^{C}\mathbf{y}_{ij}\ln\mathbf{z}_{ij} + \frac{1}{2}\sum_{\ell=1}^{L}\|\mathbf{W}^{(\ell)}\|_F^2. \tag{2.20}$$

Another regularization technique is to enforce an upper bound on the norm of the learnable parameters, called **max norm constranint**. For example, after a normal gradient update, we can normalize the weights as follows:

$$\mathbf{W}^{(\ell)} \longleftarrow c\frac{\mathbf{W}^{(\ell)}}{||\mathbf{W}^{(\ell)}||}, \tag{2.21}$$

where division is the element-wise division, and the norm $||\cdot||$ could be the L1 norm, the spectral norm, or the Frobenius norm.

If we are training a neural network, **dropout** can also be used. That is, during training, we randomly and temporarily deactivate neurons with some probability $p \in [0,1]$. Let's take the neural network in Figure 2.1 for example. If we deactivate the first neuron of the 1st hidden layer, then the formula for the 2nd hidden layer will be as follows:

$$\mathbf{h}^{(2)} = \sigma(\mathbf{W}_2\widehat{\mathbf{h}}^{(1)} + b_2) \in \mathbb{R}^5, \tag{2.22}$$

where $\widehat{\mathbf{h}}^{(1)} = [h_1^{(1)}, 0, h_3^{(1)}, h_4^{(1)}]^T$.

2.3 Fourier Transform and Convolution Theorem

In the next chapter, we will talk about graph convolution operators and Graph Convolutional Networks (GCN). Graph convolution can be considered as an extension of the

convolution operator on graph domain, so we start off by introducing Fourier transform and convolution theorem.

Definition 9 (Fourier and Inverse Fourier Transform)**.** *The Fourier transform of a function f on $\mathbb{R}^d$ is given by:*

$$\mathcal{F}(f)(\xi) = \hat{f}(\xi) = \int_{\mathbb{R}^d} f(x)e^{-2\pi i x\cdot\xi}dx, \quad \xi \in \mathbb{R}^d. \tag{2.23}$$

Its attached inversion is given by:

$$\mathcal{F}^{-1}(\hat{f})(x) = \int_{\mathbb{R}^d} \hat{f}(\xi)e^{2\pi i x\cdot\xi}d\xi, \quad x \in \mathbb{R}^d. \tag{2.24}$$

Theorem 1 (Convolution Theorem)**.** *Let f_1 and $f_2 \in L^1(\mathbb{R}^d)$. Then the Fourier transform of the convolution of two functions is the pointwise product of their Fourier transform:*

$$\mathcal{F}(f_1 * f_2) = \mathcal{F}(f_1) \odot \mathcal{F}(f_2) \tag{2.25}$$

*Therefore, the **convolution operator** $*$ can be defined as:*

$$f_1 * f_2 = \mathcal{F}^{-1}(\mathcal{F}(f_1) \odot \mathcal{F}(f_2)). \tag{2.26}$$

Chapter 3

Graph Neural Networks Architecture

In this chapter, we discuss three popular graph neural network architectures, including Graph Convolutional Network (GCN), GraphSAGE, and Graph Attention Network (GAT). Here, we consider only simple undirected graphs. We also present below the list of common used notation.

3.1 Graph Convolutional Network (GCN)

Many GNN architectures rely on convolution to propagate information. The use of convolution operators in GNN is inspired by the success of Convolutional Neural Networks (CNNs) on image and signal processing, such as ImageNet [8], or YOLOv3 [16]. There are two main approaches in designing convolution operators: spectral approach and spatial approach. Spectral approach defines the convolution operator in the spectral domain, such as ChebNet [6] while spatial approach defines convolution based only on graph topology, such as GraphSAGE [10]. GCN can be viewed as both a spectral and spatial GNN architecture.

Suppose we are given a simple undirected graph $G = (V, E)$ (where $N = |V|$ is the number of nodes, $V = \{v_1, .., v_N\}$), and a node features matrix $\mathbf{X} \in \mathbb{R}^{N \times d}$, where d is the number of features. Let $\mathbf{A} \in \mathbb{R}^{N \times N}$ be the adjacency matrix of the graph G. It's assumed in [11] that self-connections and edges to neighboring nodes are of equal importance, so we introduce $\widehat{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$, which can be regarded as the adjacency matrix with self-connections. The use of $\widehat{\mathbf{A}}$ in place of $\mathbf{A}$, which is called "renormalization trick" by the author of GCN in [11], was later confirmed to be effective, both theoretically ([21]) and empirically ([12], [21]). In [21], the authors prove that the renormalization trick shrinks the value of the largest

eigenvalue of the corresponding Laplacian matrix $\widehat{\mathbf{L}}^{sym} = \mathbf{I}_N - \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}$. According to the author, this acts as a low-pass filter which attenuate signals of higher frequency, thus produces smooth features over the graph. The result is that neighboring nodes are more likely to share similar representations and predictions.

3.1.1 Spectral Motivation

Before discussing GCN's spectral motivation, we recall some useful definitions.

Definition 10 (Graph Fourier Transform)**.** *[5, 4] Let $\widehat{\mathbf{L}}^{sym} \in \mathbb{R}^{N\times N}$ be the Laplacian matrix of a simple undirected graph G, and $\widehat{\mathbf{L}}^{sym} = \mathbf{U}^T\Lambda\mathbf{U}$ be a spectral decomposition of $\widehat{\mathbf{L}}^{sym}$. A **graph Fourier transform** associated with the graph G and $\mathbf{U}$ is $\mathcal{F}: \mathbb{R}^N \to \mathbb{R}^N$ defined as follows:*

$$\mathcal{F}(\mathbf{x}) := \mathcal{F}_{\mathbf{U}}(\mathbf{x}) = \mathbf{U}^T\mathbf{x}, \quad \forall\, \mathbf{x} \in \mathbb{R}^N, \tag{3.1}$$

*and the **graph inverse Fourier transform** associated with the graph G is $\mathcal{F}^{-1}: \mathbb{R}^N \to \mathbb{R}^N$ given by*

$$\mathcal{F}^{-1}(\mathbf{x}) := \mathcal{F}^{-1}_{\mathbf{U}}(\mathbf{x}) = \mathbf{U}\mathbf{x}, \quad \forall \mathbf{x} \in \mathbb{R}^N. \tag{3.2}$$

Definition 11 (Graph Convolution)**.** *Let $w, h \in \mathbb{R}^N$. Given graph G and its fixed and chosen eigenmatrix $\mathbf{U}$ [18], we define the **graph convolution operator** of w and h as follows:*

$$\mathbf{w} *_G \mathbf{h} := \mathcal{F}^{-1}(\mathcal{F}(\mathbf{w}) \odot \mathcal{F}(\mathbf{h})) \tag{3.3}$$

It should be noted that there may not be a unique choice for $\mathbf{U}$. However, we still assume that $\mathbf{U}$ is already chosen and fixed [18].

Lemma 1. *Let $\mathbf{w}, \mathbf{h} \in \mathbb{R}^N$. Then*

$$\mathbf{w} *_G \mathbf{h} = \mathbf{U}\mathrm{diag}(\mathbf{U}^T\mathbf{w})\mathbf{U}^T\mathbf{h} = \widehat{\mathbf{w}}(\widehat{\mathbf{L}}^{sym})\mathbf{h}$$

where $\widehat{\mathbf{w}}(\widehat{\mathbf{L}}^{sym})$ is a matrix that depends on $\mathbf{w}$, the Laplacian matrix $\widehat{\mathbf{L}}^{sym}$, and its orthogonal matrix $\mathbf{U}$.

Proof. It follows directly from the Graph Convolution definition. □

Different choices of $\mathbf{w}$ lead to different spectral GNN architectures, including GCN, ChebNet, etc. We will show how GCN can be viewed as a spectral method. It should be noted that there are some differences between our approach and the original approach that is shown in [11].

Lemma 2. *Under the settings above, if* $\mathbf{w} = c\mathbf{U}(\mathbf{1}_{N\times 1} - \boldsymbol{\lambda})$ *where* $\boldsymbol{\lambda} = [\lambda_1, \ldots, \lambda_N]^T$ *and* $c \in \mathbb{R}$*, then*

$$\mathbf{w} *_G \mathbf{h} = c\widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{h}. \tag{3.4}$$

Proof. Recall that the Laplacian matrix is defined as $\widehat{\mathbf{L}}^{sym} = \mathbf{I} - \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}$ (normalized), which has a fixed set of eigenvectors $\mathbf{U}$.

Consider $\mathbf{w} = c\mathbf{U}(\mathbf{1} - \boldsymbol{\lambda})$ where $c \in \mathbb{R}$. Then $\mathbf{U}^T\mathbf{w} = c\mathbf{U}^T\mathbf{U}(\mathbf{1} - \boldsymbol{\lambda}) = c(\mathbf{1} - \boldsymbol{\lambda}) = \mathbf{I} - \Lambda$. With $\mathbf{w}$ defined as such, we can easily verify that $\mathbf{U}^T\mathbf{w} = c(\mathbf{1} - \boldsymbol{\lambda})$, and $\text{diag}(1 - \boldsymbol{\lambda}) = \mathbf{I} - \Lambda$, where $\Lambda = \text{diag}(\boldsymbol{\lambda})$. Thus,

$$\begin{aligned}
\mathcal{C}_G(\mathbf{h}; \mathbf{w}) = \mathcal{C}_G(\mathbf{h}; c) = \mathbf{w} *_G \mathbf{h} &= \mathbf{U}(\mathbf{U}^T\mathbf{w} \odot \mathbf{U}^T\mathbf{h}) \\
&= c\mathbf{U}(\mathbf{I} - \Lambda)\mathbf{U}^T\mathbf{h} \\
&= c(\mathbf{I} - \widehat{\mathbf{L}}^{sym})\mathbf{h} \\
&= c\widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{h} \in \mathbb{R}^N.
\end{aligned}$$

□

Denote $\breve{\mathbf{A}} = \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}$. Then $\mathcal{C}(\mathbf{h}; \mathbf{w}) = c\breve{A}\mathbf{x}$.

The above equation is for the simplest case where we have only one filter and the number of channel is also one. We can generalize to more complex cases, where we may have many filters, and each of them are multi-channel.

Let's consider a scenario where our input consists of N nodes and $d = n^{(0)}$ channels. Instead of $\mathbf{x} \in \mathbb{R}^N$ and $c \in \mathbb{R}$, now we have an input matrix of $n^{(0)}$ features $\mathbf{X} \in \mathbb{R}^{N\times n^{(0)}}$ and learnable filter parameters $\mathbf{c} = [c_1, \ldots, c_{n^{(0)}}]^T \in \mathbb{R}^{n^{(0)}}$. Let us define the multi-channel graph convolutional layer operator $\mathcal{C}_G$ on $\mathbf{X}$ as belows:

$$\mathcal{C}_G(\mathbf{X}; \mathbf{c}) = \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{c} \in \mathbb{R}^N \tag{3.5}$$

Now, let's say we have $n^{(1)}$ such filters. We stack the filter parameters $\mathbf{c}^{(j)} \in \mathbb{R}^{n^{(0)}}$ where $j \in [n^{(1)}]$ horizontally to form a matrix $\mathbf{W}^{(0)} \in \mathbb{R}^{n^{(0)}\times n^{(1)}}$ to obtain:

$$\mathcal{C}_G(\mathbf{X}; \mathbf{W}^{(0)}) = \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{W}^{(0)} \in \mathbb{R}^{N\times n^{(\ell+1)}} \tag{3.6}$$

Note that Equation (3.5) and (3.6) are generalizations of the graph convolution operator between two vectors. By adding a nonlinear activation σ, we obtain the formula for the first hidden layer of GCN:

$$\mathbf{H}^{(1)} = \sigma(\mathcal{C}_G(\mathbf{X}; \mathbf{W}^{0})) = \sigma\left(\breve{\mathbf{A}}\mathbf{H}^{(0)}\mathbf{W}^{(0)}\right) \in \mathbb{R}^{N\times n^{(\ell+1)}} \tag{3.7}$$

where $\mathbf{H}^{(1)} \in \mathbb{R}^{N\times n^{(1)}}, \mathbf{H}^{(0)} = \mathbf{X} \in \mathbb{R}^{N\times n^{(0)}}$ are the output and input of the first hidden layer, respectively. It can be generalized to any layer $l \in [0, L-1]$, by letting $\mathbf{H}^{(\ell+1)}$ be input to the $(\ell+1)^{th}$ layer, $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n^{(\ell)}\times n^{(\ell+1)}}$ and stacking L such layers altogether. The **single-layer formula of GCN** is given by:

$$\mathbf{H}^{(\ell+1)} = \sigma\Big(\breve{\mathbf{A}}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}\Big) \qquad \text{where } \ell = 0, .., L-2. \tag{3.8}$$

except for the output layer, where there is no nonlinear activation:

$$\mathbf{H}^{(L)} = \breve{\mathbf{A}}\mathbf{H}^{(L-1)}\mathbf{W}^{(L-1)}. \tag{3.9}$$

In summary, the final output of a GCN with L layers is:

$$\mathbf{H}^{(L)} = \breve{\mathbf{A}}\sigma\Big(\breve{\mathbf{A}}\sigma\big(\dots\sigma(\breve{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})\dots\big)\mathbf{W}^{(L-2)}\Big)\mathbf{W}^{(L-1)}, \tag{3.10}$$

where $\breve{\mathbf{A}} = \widehat{\mathbf{D}}^{-\frac{1}{2}}\widehat{\mathbf{A}}\widehat{\mathbf{D}}^{-\frac{1}{2}}$.

3.1.2 Spatial Motivation

GCN can also be viewed as a spatial architecture. That is, at any particular layer, a node passes and receives information to all of its neighbors. The details are given below.

Lemma 3. *Let $\mathbf{h}_{v_i}^{(\ell)}$ be the ith row of $\mathbf{H}^{(\ell)}, i \in [N]$, where $\mathbf{H}^{(\ell)}$ is defined by Equation (3.8). Then we have:*

$$\mathbf{h}_{v_i}^{(\ell+1)} = \sigma\Big(\sum_{v_k\in\mathcal{N}(v_i)} \mathbf{W}^{(\ell),T}\mathbf{h}_{v_k}^{(\ell)}\Big) \in \mathbb{R}^{n^{(l+1)}}, \quad i \in [N]. \tag{3.11}$$

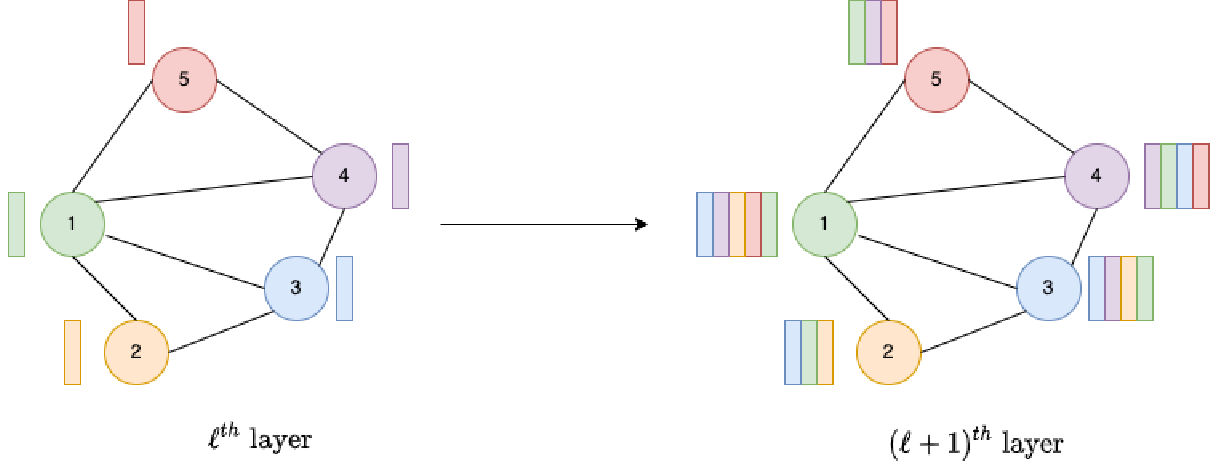
That is, a node's hidden state fcor the next layer is an aggregation of information from all its neighbors at the current layer.

Proof. Denote $\mathbf{B} = \breve{\mathbf{A}}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}$ and $h_{kj}^{(\ell)}$ the jth entry of $\mathbf{h}_{v_k}^{(\ell)}$. It is sufficient to prove that $B_{ij} = \left(\sum_{v_k\in\mathcal{N}(v_i)} \mathbf{W}^{(\ell),T}\mathbf{h}_{v_k}^{(\ell)}\right)_j$, for $i \in [N]$ and $j \in [n^{(l+1)}]$.

Since $\breve{a}_{ik} = 1$ if $v_k \in \mathcal{N}(v_i)$ and $\breve{a}_{ik} = 0$ if $v_k \notin \mathcal{N}(v_i)$, for $i \in [N], j \in [n^{(\ell+1)}]$, we have:

$$(\breve{\mathbf{A}}\mathbf{H}^{(l)})_{ij} = \Big(\sum_{k=1}^{N} \breve{a}_{ik}h_{kj}^{(\ell)}\Big) = \Big(\sum_{v_k\in\mathcal{N}(v_i)} h_{kj}^{(\ell)}\Big).$$

Figure 3.1: GCN as a spatial architecture - information is passed from a node to its direct neighbors. Information of node 5 at ℓ^{th} layer is passed to nodes 1 and 4 at $(\ell+1)^{th}$ layer. The hidden representation of node 2 at $(\ell+1)^{th}$ layer is calculated based on the hidden representation of nodes 1 and 3 at ℓ^{th} layer.



Therefore,

$$B_{ij} = (\breve{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(\ell)})_{ij} = \sum_{p=1}^{N}(\breve{\mathbf{A}}\mathbf{H}^{(\ell)})_{ip}w_{pj}^{(\ell)} = \sum_{p=1}^{N}\Big(\sum_{v_k\in\mathcal{N}(v_i)} h_{kp}^{(\ell)}\Big)w_{pj}^{(\ell)}. \tag{3.12}$$

On the other hand,

$$\sum_{v_k\in\mathcal{N}(v_i)} \mathbf{W}^{(\ell),T}\mathbf{h}_{v_k}^{(\ell)} = \sum_{v_k\in\mathcal{N}(v_i)} \Big[\sum_{p=1}^{N} h_{kp}^{(\ell)}w_{p1}^{(\ell)} \dots \sum_{p=1}^{N} h_{kp}^{(\ell)}w_{pN}^{(\ell)}\Big]^T. \tag{3.13}$$

Hence the jth element of Equation (3.13) is

$$\Big(\sum_{v_k\in\mathcal{N}(v_i)} \mathbf{W}^{(\ell)^T}\mathbf{h}_k^{(\ell)}\Big)_j = \sum_{v_k\in\mathcal{N}(v_i)}\sum_{p=1}^{N} h_{kp}^{(\ell)}w_{pj}^{(\ell)} = \sum_{p=1}^{N}\Big(\sum_{v_k\in\mathcal{N}(v_i)} h_{kp}^{(\ell)}\Big)w_{pj}^{(\ell)},$$

which completes the proof. □

3.1.3 Efficient Implementation

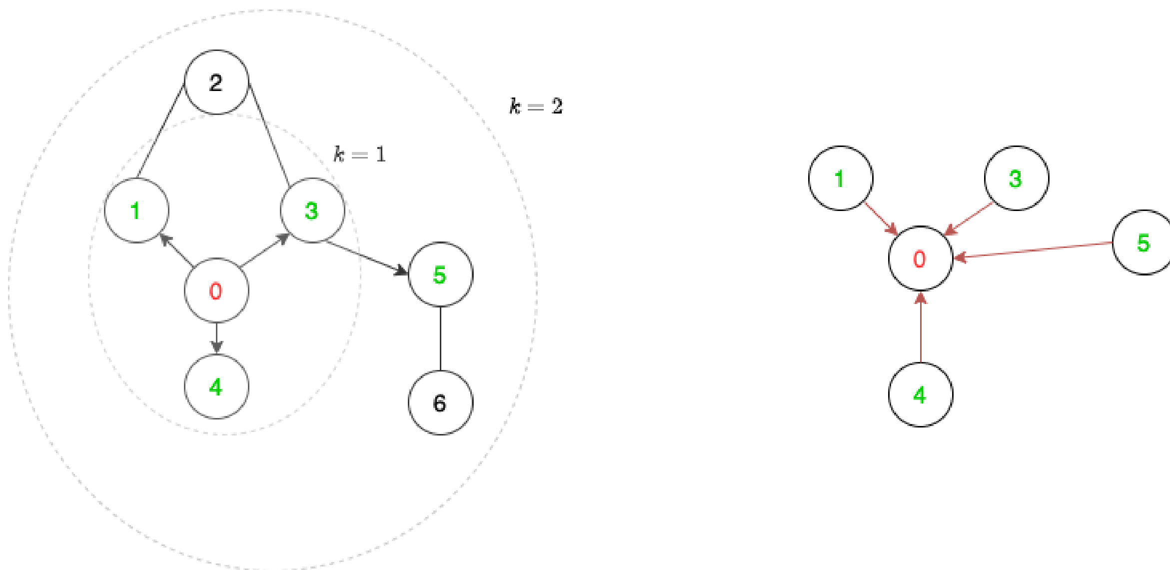
In previous sections, we see that GCN can be implemented both in graph level as in (3.8) and node level as in (3.11). In this section, we discuss the pros and cos of each implementation.

The graph-level implementation as in (3.8) should be used when our priority is to minimize the number of mathematical operations. It is because we only compute the embedding $\mathbf{h}_{v_i}^{(\ell)}$ exactly once [9]. However, this implementation requires the full adjacency matrix and node features matrix simultaneously. Therefore, it should not be used if memory is our top concern. Moreover, using graph-level implementation means that we cannot take advantage of mini-batching.

On the node-level implementation should be considered when we are limited in memory, or when the input graph is big. We can also use mini-batching to further control the memory footprint of the GNN.

3.2 GraphSAGE

Figure 3.2: Visualization of GraphSAGE. **Left**: Sampled neighborhood of node 0, $\mathcal{N}_0 = \{1, 3, 4, 5\}$. **Right**: Node 0 aggregates information from its neighborhood.



One drawback of GCN is that its training heavily depends on the adjacency matrix $\mathbf{A}$. In transductive settings where all nodes are given at first, GCN can work fairly well. However, in inductive settings where a new node or edge is introduced to the graph, we must retrain the GCN from scratch. On the other hand, GraphSAGE is an architecture that does not require the adjacency matrix $\mathbf{A}$, so it works well in both transductive and inductive settings [10].

Moreover, GraphSAGE presents a novel sampling approach that is useful for processing very large graphs. In such large graphs, the number of nodes and the size of the neighborhoods are big, thus it is difficult to store and process all the neighborhood information. To deal with that, GraphSAGE selects only a subset of each node's neighborhood, ensuring the size of the neighborhood is reasonble for message passing. [24] [10].

Suppose we are given a graph G of N vertices, and a matrix of node features $\mathbf{X} \in \mathbb{R}^{N \times d}$. Let $\mathbf{h}_u^{(\ell)} \in \mathbb{R}^{n^{(\ell)}}$ be the representation of node u at layer ℓ, where $\mathcal{N}_s(v)$ is the sampled neighborhood of node v. We stack them vertically to obtain $\mathbf{H}^{(\ell)} = [\mathbf{h}_1^{(\ell)^T}, .., \mathbf{h}_N^{(\ell)^T}] \in \mathbb{R}^{N \times n^{(\ell)}}$, which is the output of the $\ell-$th layer ($\ell \in [L]$). Let $\mathcal{G}_\ell$ be a differentiable aggregate function, at layer ℓ. Let's also denote $\mathbf{h}_{\mathcal{N}_s(v)}^{(\ell)} \in \mathbb{R}^{n^{(\ell-1)}}$ as the aggregated message from neighborhood of node v, at layer ℓ. Then the formula for $\mathbf{h}_{\mathcal{N}_s(v)}^{(\ell)}$ is as follows:

$$\mathbf{h}_{\mathcal{N}_s(v)}^{(\ell)} = \mathcal{G}_\ell\Big(\{\mathbf{h}_u^{(\ell-1)}, \forall u \in \mathcal{N}_s(v)\}\Big) \tag{3.14}$$

There can be many choices for $\mathcal{G}$, such as mean, pooling, or long-short term memory aggregators [10]. For example, if $\mathcal{G}$ mean aggregator:

$$\mathbf{h}_{\mathcal{N}_s(v)}^{(l)} = \sigma\Big(\mathbf{W}\mathcal{M}(\mathbf{h}_u^{(\ell-1)}, \forall u \in \mathcal{N}_s(v) \cup \{v\}\Big), \tag{3.15}$$

where $\mathbf{W} \in \mathbb{R}^{n^{(\ell)} \times n^{(\ell-1)}}$, $\mathcal{M}$ is the mean operator, $\mathcal{M}(\mathbf{a}_1, .., \mathbf{a}_k) = \frac{1}{k}\sum_{i=1}^{k} \mathbf{a}_i \in \mathbb{R}^p$ for $\mathbf{a}_i \in \mathbb{R}^p$. If $\mathcal{N}_s(v)$ is further assumed as the set of actual neighboring nodes of v, which is $\mathcal{N}(v)$, then this is equivalent to Equation (3.11).

The operator $\mathcal{G}$ can also be a pooling aggregator:

$$\mathbf{h}_{\mathcal{N}_s(v)}^{(l)} = \max\{\sigma(\mathbf{W}\mathbf{h}_u^{(\ell-1)} + b), \forall u \in \mathcal{N}_s(v)\}, \tag{3.16}$$

where $\mathbf{W} \in \mathbb{R}^{n^{(\ell)} \times n^{(\ell-1)}}$ and max is an element-wise operation:

$\max(\mathbf{a}_1, .., \mathbf{a}_k) = [\max(a_{11}, .., a_{k1}), .., \max(a_{1p}, .., a_{kp})] \in \mathbb{R}^p$ where $\mathbf{a}_1, .., \mathbf{a}_k \in \mathbb{R}^p$.

After choosing an appropriate aggregator for $\mathbf{h}^{(\ell)}_{\mathcal{N}_s(v)}$, a single-layer formula of GraphSAGE is defined as follow:

$$\mathbf{h}^{(\ell)}_v = \sigma\Big(\mathbf{W}^{(\ell)}[\mathbf{h}^{(\ell-1)}_v||\mathbf{h}^{(\ell)}_{\mathcal{N}_s(v)}]\Big) \tag{3.17}$$

where $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n^{(\ell)}\times 2n^{(\ell-1)}}$.

In summary, the output embedding for a node $v \in V$ of a GraphSAGE architecture with two layers is given by:

$$\mathbf{h}^{(1)}_{\mathcal{N}_s(v)} = \mathcal{G}_1\Big(\{\mathbf{h}^{(0)}_u, \forall u \in \mathcal{N}_s(v)\}\Big) \tag{3.18}$$

$$\mathbf{h}^{(1)}_v = \sigma\Big(\mathbf{W}^{(1)}[\mathbf{h}^{(0)}_v||\mathbf{h}^{(1)}_{\mathcal{N}_s(v)}]\Big) \tag{3.19}$$

$$\mathbf{h}^{(2)}_{\mathcal{N}_s(v)} = \mathcal{G}_2\Big(\{\mathbf{h}^{(1)}_u, \forall u \in \mathcal{N}_s(v)\}\Big) \tag{3.20}$$

$$\mathbf{h}^{(2)}_v = \mathbf{W}^{(2)}[\mathbf{h}^{(1)}_v||\mathbf{h}^{(2)}_{\mathcal{N}_s(v)}] \tag{3.21}$$

where $v \in V$, $h^{(0)}_u$ is the input features of node u, and $h^{(2)}_v$ is the final embedding output of node v.

When looking throughout the above equations, we can see that: given a new node v, we can induce its embedding with only the embeddings of its neighbors. The entire graph structure is not required, so we do not need to retrain for a new node. This is not possible if we use Equation (3.8) to implement GCN.

3.3 Graph Attention Network (GAT)

From Equation (3.11), we can see that GCN treats information from all nodes in neighborhood equally. More precisely, embeddings of all nodes v_j in neighborhood of v_i are multiplied by the same matrix $\mathbf{W}^{(\ell)}$. Graph Attention Network (GAT) [20] takes a different approach; it adapts attention mechanism to assign and learn different weights for different neighbors (see Equation (3.25)).

For each $u \in V = \{v_1, \dots, v_N\}, \ell \in [L]$, let $\mathbf{h}^{(\ell)}_u \in \mathbb{R}^{n^{(\ell)}}$ be the representation of node u at layer ℓ and $\mathbf{H}^{(\ell)} \in \mathbb{R}^{N\times n^{(\ell)}}$ be the matrix whose rows are $\mathbf{h}^{(\ell)}_i, i \in [N]$. Let $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n^{(\ell+1)}\times n^{(\ell)}}$

Figure 3.3: GAT: From hidden representation to attention weights (See Equations (3.22) - (3.24))

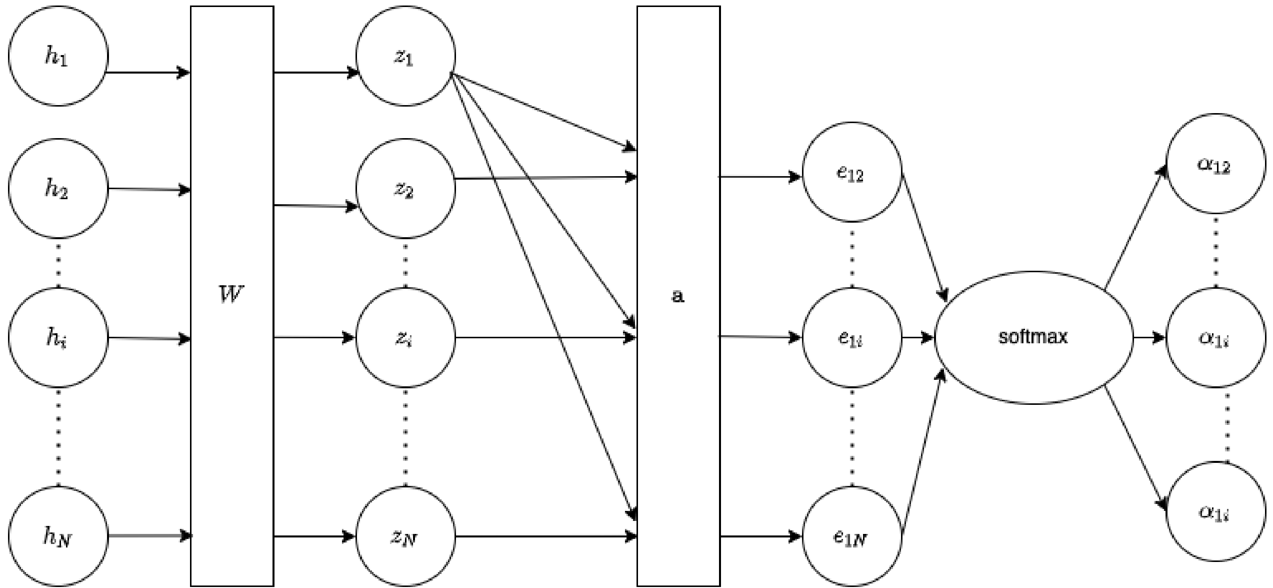
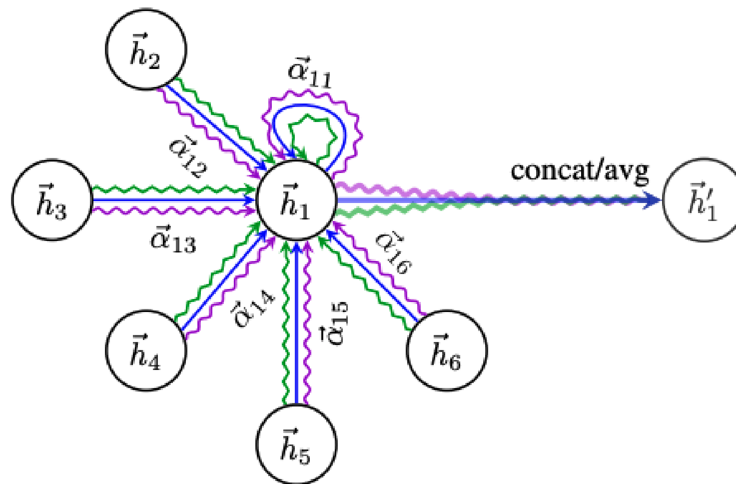


Figure 3.4: An illustration of multihead attention (with K = 3 heads) by node 1 on its neighborhood. Different colors represent different heads. The aggregated features from each head are concatenated or averaged to obtain h'_1, which is the representation of node 1 at the next layer. The figure is taken from [20].



be the shared weight matrix that is applied for every node, at the ℓ-th layer. First of all, the embedding of node u at layer ℓ is transformed linearly:

$$\mathbf{z}_u^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{h}_u^{(\ell)} \in \mathbb{R}^{n^{(\ell+1)}}. \tag{3.22}$$

Let $\mathbf{a}_\ell \in \mathbb{R}^{2n^{(\ell+1)}}$, the attention mechanism, be a learnable sing-layer feedforward neural network. The unnormalized attention score of node u to node v, denoted as $e_{uv}^{(\ell)}$, is defined as:

$$e_{uv}^{(\ell)} = \text{LeakyReLU}(\mathbf{a}^{(\ell),T}[\mathbf{z}_u^{(\ell)}||\mathbf{z}_v^{(\ell)}]) \in \mathbb{R}. \tag{3.23}$$

The normalized attention of node u to node v (or the importance of node v to node u) at layer l, denoted as $\alpha_{uv}^{(\ell)} \in \mathbb{R}$, is defined as:

$$\alpha_{uv}^{(\ell)} = \text{softmax}(e_{uv}^{(\ell)}) = \frac{\exp(e_{uv}^{(\ell)})}{\sum\limits_{w \in \mathcal{N}(u)} \exp(e_{uw}^{(\ell)})} \in \mathbb{R}. \tag{3.24}$$

Lastly, the embedding of node u at layer $\ell + 1$ is:

$$\mathbf{h}_u^{(\ell+1)} = \sigma\Big(\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^{(\ell)}\mathbf{z}_v^{(\ell)}\Big) \in \mathbb{R}^{n^{(\ell+1)}}. \tag{3.25}$$

In practice, multiple such attention weights are used simultaneously to stabilize the learning process [20]. This technique is called multi-head attention, where (3.25) are applied K times using K different learnable weight matrices. These K outputs are then concatenated or averaged as follows:

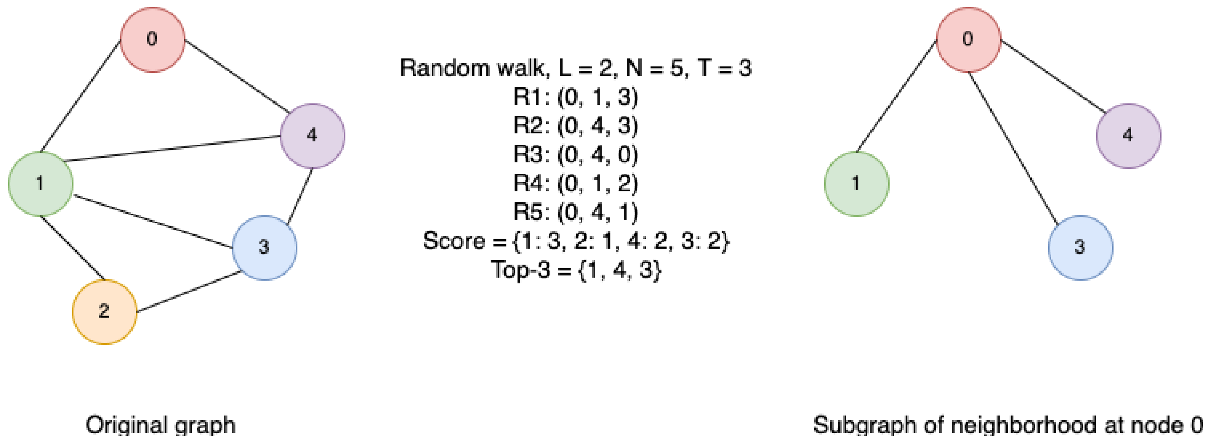
$$\mathbf{h}_u^{(\ell+1)} = ||_{k=1}^{K}\sigma\Big(\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^{(\ell),k}\mathbf{W}_k^{(\ell)}\mathbf{h}_v^{(\ell)}\Big) \in \mathbb{R}^{n^{(\ell+1)}}, \tag{3.26}$$

where K is the number of attention heads, $\mathbf{W}_k^{(\ell)} \in \mathbb{R}^{\frac{n^{(\ell+1)}}{K} \times n^{(\ell)}}$ is the weight matrix for the kth attention head, at ℓ^{th} layer, $k \in [K]$.

3.4 PinSAGE

Similar to GraphSAGE, PinSAGE also utilizes the sampling technique for reducing the size of the neighborhoods. The difference is that while GraphSAGE uses random sampling,

Figure 3.5: Visualization of PinSAGE's importance-based neighborhood sampling. **Left**: Original graph **Center**: Simulation of $N = 5$ random walks from node 0, each with length $L = 2$, top $T = 3$ is chosen. **Right**: Node 0 aggregates information from its sampled neighborhood.



PinSAGE proposes an importance-based sampling method. For a given target node u, it performs random walks starting from u, count the occurrences of the nodes appearing on each walk, then finally choose the top T nodes with the highest visit counts. Having a fixed size of neighborhoods helps us control the memory footprint for our algorithm.

Suppose we are given a graph G of N vertices, and a matrix of node features $\mathbf{X} \in \mathbb{R}^{N \times d}$. Let $\mathbf{h}_u^{(\ell)} \in \mathbb{R}^{n^{(\ell)}}$ be the representation of node u at layer ℓ. Let $\mathcal{G}_\ell$ be a differentiable aggregate function, at layer ℓ. For example, $\mathcal{G}_\ell$ can be mean/LSTM/pooling. Let's also denote $\mathbf{h}_{\mathcal{N}(v)}^{(\ell)} \in \mathbb{R}^{n^{(\ell)}}$ as the aggregated message from neighborhood of node v, at layer ℓ.

Let $T = |\mathcal{N}(v_1)| = .. = |\mathcal{N}(v_i)| = ... = |\mathcal{N}(v_N)| \, \forall i \in [N]$ be the fixed size of neighborhoods for every node. $\mathcal{N}(u)$ of size T and $\alpha(u) \in \mathbb{R}^T$ for are the set of neighboring nodes, and set of neighbor weights of node u, respectively. We also use $\gamma : \mathbb{R}^{n^{(\ell)} \times T} \to \mathbb{R}^{n^{(\ell)}}$ to denote a symmetric vector function. For example, γ can be element-wise mean or weighted sum. $\mathbf{Q}^{(\ell)} \in \mathbb{R}^{n^{(\ell)} \times n^{(\ell)}}, \mathbf{W}^{(\ell)} \in \mathbb{R}^{n^{(\ell+1)} \times 2n^{(\ell)}}$ be learnable weight matrices, $\mathbf{q} \in \mathbb{R}^{n^{(\ell)}}, \mathbf{w} \in \mathbb{R}^{n^{(\ell)}}$ be learnable bias vectors. The aggregated message from neighborhood of node u, $h_{\mathcal{N}(u)}^{(\ell)}$ is computed as follows:

$$h_{\mathcal{N}(u)}^{(\ell)} = \gamma\Big(\{\sigma(\mathbf{Q}^{(\ell)} h_v^{(\ell)} + \mathbf{q}) | v \in \mathcal{N}(u)\}, \alpha(u)\Big) \tag{3.27}$$

The unnormalized vector representation of node u at layer $\ell + 1$ is as follows:

$$h_{u,un}^{(\ell+1)} = \sigma\left(\mathbf{W}^{(\ell)} \cdot \left[\frac{h_u^{(\ell)}}{h_{\mathcal{N}(u)}^{(\ell)}}\right] + \mathbf{w}\right) \tag{3.28}$$

The final, normalized vector representation of node u at layer ℓ is as follows:

$$h_u^{(\ell+1)} = \frac{h_{u,un}^{(\ell+1)}}{||h_{u,un}^{(\ell+1)}||_2} \tag{3.29}$$

By looking at the above equations, we can see PinSAGE: 1) controls the memory footprint of the algorithm during training, since we chose a fixed size T for a node's neighborhood, and 2) takes into account the importance of neighbors when aggregating their representations. By performing random walks, not only we select top T nodes as neighboring nodes of u, we are able to find their importance (the corresponding element of $\alpha(u)$) to u as well.

Chapter 4

GNN Training

Graph Neural Network aims to extend existing neural networks to process graph input data. Beside what are similar to a general neural network training framework, in this chapter, we will go into specific details of several aspects of training a Graph Neural Network.

4.1 Backpropagation

Backpropagation is an algorithm based on gradient descend to optimize the parameters in a model. Backpropagation in a neural network consists of two steps: forward calculation and backward propagation. Given a set of parameters and input, forward calculation computes the values at each neuron in forward order. On the other hand, backward propagation computes the error at each variable, and update the parameters with the corresponding partial derivatives in backward order.

4.1.1 Backpropagation on GCNs

In this section, we demonstrate how backpropagation works for GCN. Consider a graph $G = (V, E)$, where $V = (v_1, \dots, v_N)$. Denote $\mathbf{Z} = \mathbf{H}^{(2)} \in \mathbb{R}^{N \times n^{(2)}}$ the output of the GCN whose i^{th} row $\mathbf{z}_{v_i} = [z_{i,1}, \dots, z_{i,n^{(2)}}]^T \in \mathbb{R}^{n^{(2)}}$ is the output vector of node v_i. Similarly, let $\mathbf{h}_{v_i}^{(\ell)} = [h_{i,1}^{(\ell)}, \dots, h_{i,n^{(\ell)}}^{(\ell)}]^T \in \mathbb{R}^{n^{(\ell)}}$ be the embedding vector at layer ℓ of node v_i. We start from Equation (3.8):

$$\mathbf{Z} = \mathbf{H}^{(2)} = \breve{\mathbf{A}}\mathbf{H}^{(1)}\mathbf{W}^{(1)} \in \mathbb{R}^{N \times n^{(2)}}, \tag{4.1}$$

$$\mathbf{H}^{(1)} = \sigma\Big(\breve{\mathbf{A}}\mathbf{H}^{(0)}\mathbf{W}^{(0)}\Big) \in \mathbb{R}^{N\times n^{(1)}}, \tag{4.2}$$

where $\breve{\mathbf{A}} \in \mathbb{R}^{N\times N}$ is the normalized adjacent matrix with self-connections of the graph G, while $\mathbf{W}^{(0)} \in \mathbb{R}^{n^{(0)}\times n^{(1)}}$ and $\mathbf{W}^{(1)} \in \mathbb{R}^{n^{(1)}\times n^{(2)}}$ are the trainable parameter matrices. Let's consider the entry at row $i \in [N]$, column $k \in [n^{(2)}]$ of $\mathbf{Z}$:

$$z_{ik} = h_{ik}^{(2)} = \sum_{j\in[N],\, p\in[n^{(0)}]} a_{ij} h_{jp}^{(1)} w_{pk}^{(1)}, \tag{4.3}$$

We recall from Equation (3.11) that the formula for $h_{jp}^{(1)}$ where $j \in [N], p \in [n^{(1)}]$ is given by:

$$h_{jp}^{(1)} = \sigma\Big(\sum_{v_q\in\mathcal{N}(v_j)} \sum_{r=1}^{n^{(0)}} h_{qr}^{(0)} w_{rp}^{(0)}\Big), \tag{4.4}$$

From Equation (4.3), the partial derivative of z_{ik} with respect to $w_{ab}^{(1)}$ is as follows:

$$\frac{\partial z_{ik}}{\partial w_{ab}^{(1)}} = \delta_{bk} \sum_{j\in\mathcal{N}(v_i)} h_{ja}^{(1)}. \tag{4.5}$$

To compute partial derivative of z_{ik} with respect to $w_{ab}^{(0)}$, we use chain rule as follows:

$$\frac{\partial z_{ik}}{\partial w_{ab}^{(0)}} = \sum_{j,p} \frac{\partial z_{ik}}{\partial h_{jp}^{(1)}} \frac{\partial h_{jp}^{(1)}}{\partial w_{ab}^{(0)}}, \tag{4.6}$$

where the partial derivative of of z_{ik} with respect to $h_{jp}^{(1)}$ is:

$$\frac{\partial z_{ik}}{\partial h_{jp}^{(1)}} = \begin{cases} w_{pk}^{(1)}, \text{ if } j \in \mathcal{N}(i) \\ 0, \text{ otherwise,} \end{cases}$$

while the partial derivative of $h_{jp}^{(1)}$ with respect to $w_{ab}^{(0)}$ depends on the choice of σ. We discuss two popular choices, including ReLU and tanh. If σ is ReLU then:

$$\frac{\partial h_{jp}^{(1)}}{\partial w_{ab}^{(0)}} = \begin{cases} \delta_{bp} \sum_{q\in\mathcal{N}(j)} h_{qa}^{(0)}, \text{ if } \Big(\sum_{q\in\mathcal{N}(j)} \sum_{r=1}^{n^{(0)}} h_{qr}^{(0)} w_{rp}^{(0)}\Big) > 0 \\ 0, \text{ otherwise.} \end{cases}$$

Thus

$$\frac{\partial z_{ik}}{\partial w_{ab}^{(0)}} = \begin{cases} \sum_{j\in\mathcal{N}(i)} w_{bk}^{(1)} \sum_{q\in\mathcal{N}(j), B(j,b)>0} h_{qa}^{(0)} \\ 0, \text{ otherwise,} \end{cases}$$

Figure 4.1: Effect of adding skip connections in graph neural networks: adding a skip connection (blue arrow) only helps mitigate the vanishing gradient risk from the node itself. The risk of vanishing gradient from message passing through its neighbors still remains. Figure is taken from [15]

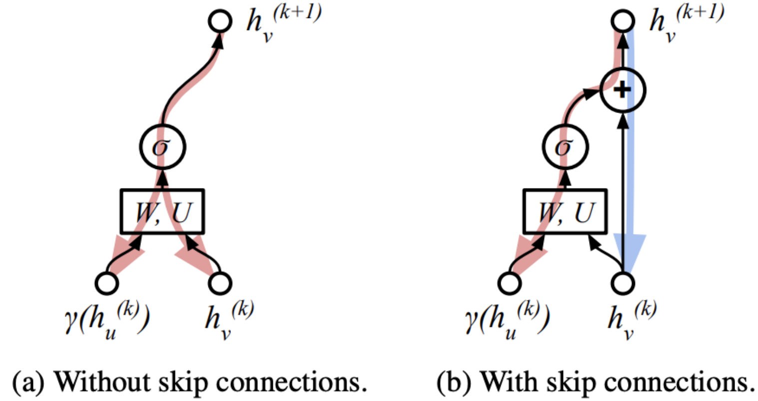


Figure 1. Effect of skip connections.

where

$$B(j,b) := \begin{cases} 1, & \text{if } \sum_{q\in\mathcal{N}(j)} \sum_{r=1}^{n^{(0)}} h_{qr}^{(0)} W_{rb}^{(0)} > 0 \\ 0, & \text{otherwise.} \end{cases}$$

If σ is tanh then:

$$\frac{\partial h_{jp}^{(1)}}{\partial W_{ab}^{(0)}} = \delta_{bp}(1 - h_{jp}^{(1)^2}) \sum_{q\in\mathcal{N}(j)} h_{qa}^{(0)}.$$

Thus

$$\frac{\partial z_{ik}}{\partial W_{ab}^{(0)}} = \delta_{kb} \sum_{j\in\mathcal{N}(i)} (1 - h_{jb}^{(1)^2}) W_{bk}^{(1)} \sum_{q\in\mathcal{N}(j)} h_{qa}^{(0)}. \tag{4.7}$$

4.1.2 Skip Connections in GNNs

Similar to a feedforward neural network, a graph neural network can also be prone to the vanishing gradient problem. There are some approaches to mitigate the effect of these problems, among those adding skip connections is one of the most popular choice. However, for graph neural networks, it is shown in [15] that while skip-connections can improve depth-wise backpropagation between the representations of same nod in successive layers, they

do not improve the breadth-wise backpropagation between representations of neighboring nodes. Moreover, it is also shown in the paper that adding skip connections in GNNs may hurt performance when the task requires learning patterns containing distant nodes. Therefore, one must be careful when considering adding skip connections to a graph neural network.

4.2 Loss Functions

There are two main classification problems on graphs, which are **node classification** and **graph classification**. In node classification, we are interested in assigning a class for each unlabelled node in the graph based on the graph information and the relations of the target node with its neighbor. For example, in a citation network, we are given a graph of research papers which are linked to each other via citationships. We need to categorize the unlabelled papers into different groups. On the other hand, in the graph-focused classification, we treat each graph as a single data. An example of graph classification is to classify images into different classes where we consider each image as a graph itself. For both two classification tasks, cross-entropy loss is frequently used.

Node classification and graph classification are similar in the sense that we seek to find an embedding vector representation for the example that we want to classify. In node classification, an example corresponds to one node, while in graph classification, an example corresponds to a graph.

We consider a general classification task, where we have N training examples $\mathcal{E} = \{E_1, \ldots, E_N\}$. Let $\mathbf{z}_i = \{z_{i1}, \ldots, z_{iC}\}$ be the output embedding vector of the ith example, and $\mathbf{y}_i = [y_{i1}, \ldots, y_{iC}]$ be the one-hot vector indicating the ground-truth class of ith example, where C is the number of classes. For example, if the true class of the ith example is k then $\mathbf{y}_i = [y_{i1}, \ldots y_{iC}] \in \mathbb{R}^C$ where:

$$\mathbf{y}_{uj} = \begin{cases} 1 \text{ if } j = k \\ 0 \text{ otherwise.} \end{cases}$$

Then the cross-entropy loss for this classfication problem is calculated as follows:

$$\mathcal{L} = -\sum_{i=1}^{N}\sum_{j=1}^{C} \mathbf{y}_{ij} \ln \mathbf{z}_{ij} \tag{4.8}$$

In particular, in a node classification problem, an example E_i corresponds to node $u_i \in V$. The output embedding vectors for all nodes are given by an embedding matrix:

$$\mathbf{Z} = \text{softmax}(\mathbf{H}^{(L)}) = [\mathbf{z}_1, .., \mathbf{z}_N]^T \in \mathbb{R}^{N \times C}$$

Then cross-entropy loss for the node classification problem is given by:

$$\mathcal{L} = -\sum_{u=1}^{N}\sum_{i=1}^{C} \mathbf{y}_{ui} \ln \mathbf{z}_{ui} \tag{4.9}$$

In a graph classification problem, an example E_i now corresponds to a graph $G_i \in \mathcal{T}$, where $\mathcal{T} = \{G_1, \ldots, G_n\}$ is the set of labeled training graphs, and $\mathbf{z}_{G_i} \in \mathbb{R}^d, \mathbf{y}_{G_i} \in \mathbb{R}^C$ are embedding output and ground-truth class of the ith graph, correspondingly. The cross entropy loss for the graph classification task is given by:

$$\mathcal{L} = -\sum_{j=1}^{N}\sum_{i=1}^{C} \mathbf{y}_{G_i} \ln \mathbf{z}_{G_i}. \tag{4.10}$$

Another popular learning problem on graphs is **link prediction**. For example, graph-based recommender systems considers a graph where users and products are to types of nodes. By utilizing the similarity among different users and different products, and the relation between users and products, the recommender make a prediction on which items a user will likely buy next, thus recommend these items to the user. There are two popular choices for the associated loss function, which are the cross-entropy loss and the max-margin loss. Both are often used with negative sampling [9].

Suppose we are given a simple undirected graph $G = (V, E)$. Denote $z_u \in \mathbb{R}^{n^{(L)}}$ as the embedding at the final layer of node u. We define the decoder function $D : V \times V \to \mathbb{R}$ as follows:

$$D(u, v) = z_u^T z_v. \tag{4.11}$$

The cross entropy loss with negative sampling for the link prediction problem is given by

$$\mathcal{L} = \sum_{(u,v)\in E} -\log(\sigma(D(z_u, z_v))) - \gamma \mathbf{E}_{v_n \sim P_{n,v}(\mathcal{V})}[\log(\sigma(-D(z_u, z_{v_n})))] \tag{4.12}$$

where σ is the logistic function, $P_{n,u}(\mathcal{V})$ denotes a "negative sampling" distribution over the set of nodes $\mathcal{V}$ which might depend on u, and $\gamma > 0$ is a hyperparameter.

The above formula can be decomposed into two components. The first component, $\log(\sigma(D(z_u, z_v)))$, equals the log-likelihood that we predict "true" for an actual edge.

The second component, $\mathbf{E}_{v_n \sim P_{n,v}(\mathcal{V})}[\log(\sigma(-D(z_u, z_{v_n})))]$, equals the expected log-likelihood that we correctly predict "false" for an false edge (edge that does not exist in graph). In practice, the expectation is evaluated using a Monte Carlo approximation and the most popular form of this loss is

$$\mathcal{L} = \sum_{(u,v)\in E} -\log(\sigma(D(z_u, z_v))) - \sum_{v_n \in \mathcal{P}_{n,u}} [\log(\sigma(-D(z_u, z_{v_n})))], \tag{4.13}$$

where $\mathcal{P}_{n,u}$ is a small set of nodes sampled from $P_{n,v}(\mathcal{V})$.

The **max-margin loss** (sometimes called **hinge loss**) is calculated as follows:

$$\mathcal{L} = \sum_{u,v\in E} \sum_{v_n \in \mathcal{P}_{n,u}} \max(0, -D(z_u, z_v) + D(z_u, z_{v_n}) + \Delta), \tag{4.14}$$

where $\Delta > 0$. Max-margin loss compares the direct output of the decoders. If the score for the true pair is bigger than the false pair, we have small loss. Δ is called the margin, and the loss for a pair will equal 0 if the difference in scores is at least Δ.

4.3 Special Training Methods on Graphs

4.3.1 Mini-Batching

In order to control the memory footprint of a GNN, we can use a technique called **mini-batching**. The idea is to run the node-level message passing equations for a subset of nodes in the graph in each batch. Redundant computations can be avoided through careful engineering to ensure that we only compute the embedding $\mathbf{h}_u^{(\ell)}$ for each node u in the batch at most once.

However, this technique has one significant limitation. That is, we cannot simply run message passing on a subgraph withotu losing information. Every time we remove a node, we also remove its edges. Therefore, it is possible that two nodes u, v in the subgraph are no longer connected, even if they are connected in the full graph. Several strategies have been proposed to overcome this limitation, including subsampling node neighborhoods of each node.

4.3.2 Sampling

Simply using the full graph for training is expensive, especially for dense graphs. Moreover, since each graph is different, adding unseen nodes to a graph would require starting training

from scratch. To address these issues, several GNN architecture use different sampling methods to sample a subset of nodes for message passing. There are many different ways to perform sampling on graphs, most of them can be put into one of three categories: node sampling, layer sampling, and subgraph sampling.

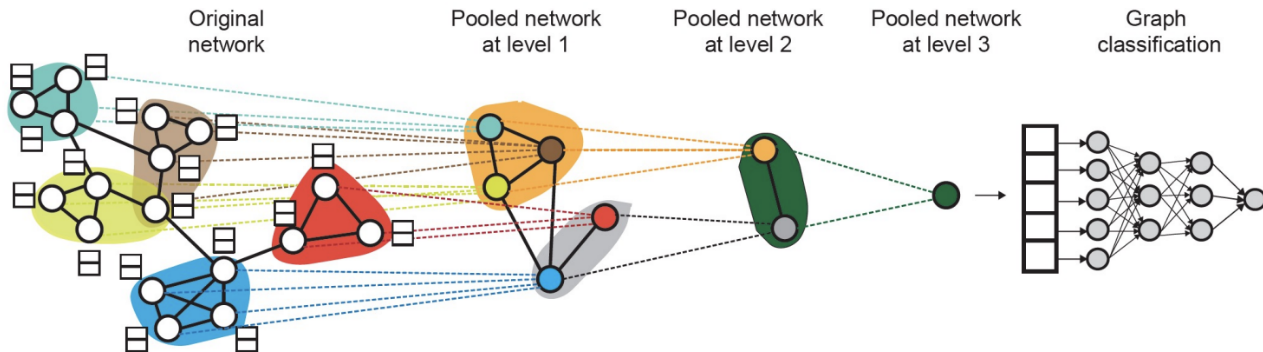
Node sampling is to sample a fixed small number of neighbors for each node. For example, as we can see from (3.20), **GraphSAGE** performs neighborhood sampling/ and aggregation of embeddings on the sampled nodes. This keeps the computational and memory complexity constant with respect to the size of the graph.

PinSAGE is an extension of GraphSAGE on large graphs. Instead of random sampling, it uses an importance-based sampling approach. Importance of a node v to node u is proportion to the number of times v appears in a random walk from u. Finally, we may simply select top T nodes with highest importance score to u, or sampling T nodes with probability proportional to the visit count of each node. This importance-based approach has a interesting property: a node v which is not a direct neighbor of u can be more important to u than some of its direct neighbors. Thus, the sampled neighborhood here is not just a subset of the "traditional" neighborhood of u which we can deduce from adjacency matrix $\mathbf{A}$.

While node sampling methods sample neighbors for each node, **layer sampling** methods chooses a small set of nodes for aggregation in each layer to control the expansion factor. That means the number of neighbors for message aggregation of each node can be different. **FastGCN (Chen et al., 2018b)** directly samples the receptive field for each layer. It uses importance sampling, where the important nodes are more likely to be sampled. In contrast to fixed sampling methods above, Huang et al. (2018) introduce a parameterized and trainable sampler to perform layer-wise sampling conditioned on the former layer. Furthermore, this adaptive sampler could optimize the sampling importance and reduce variance simultaneously. LADIES (Zou et al., 2019) intends to alleviate the sparsity issue in layer-wise sampling by generating samples from the union of neighbors of the nodes.

Meanwhile, **subgraph sampling** methods sample multigraphs first, then perform neighborhood search within the sampled subgraphs. ClusterGCN (Chiang et al., 2019) samples subgraphs by graph clustering algorithms, while GraphSAINT (Zeng et al., 2020) directly samples nodes or edges to generate a subgraph.

Figure 4.2: DiffPool [23] for graph classifcation problems. At each hierarchical layer, a GNN is run to obtain embeddings of nodes. The learn learned embeddings are used to cluster nodes together. Then another GNN layer is run on this coarsened graph. This whole process is repeated for L layers and the final output representation is used to classify the graph.



4.3.3 Graph Pooling

In convolutional neural networks (CNNs), there are usually pooling layers following a convolution layers. The idea of pooling layers is to get more general features. Due to the great success of CNNs in dealing with images, there has been a lot of effort on extending pooling modules to graph structures.

As a direct extension from CNNs, the node-wise aggregation operators, such as max, mean, sum, are still some of the most popular choices for designing graph pooling modules. Apart from that, there are hierarchical pooling modules which utilizes the hierarchical property of the graph structure, including DiffPool [23]. The idea of **DiffPool** is to train two GNNs (A, B) in parallel, at each level. GNN A computes node embeddings for each node, GNN B computes the cluster that a node belongs to. A uses clustering assignments from B to aggregate node embeddings for each cluster. These node embeddings are then used as input for the next pooling layer.

4.3.4 Augmentation

Augmentation is the process of adding features, nodes, or edges to enrich the graph. When the input graph does not have features, we can assign unique IDs (one-hot vectors) to nodes. Sometimes there are extra properties that may be useful such as node degrees,

cycle counts, or clustering coefficients, we can add them manually as well. On the other hand, graph augmentation is to add nodes or edges to graphs. When a graph is sparse the message passing process may be difficult. In that case, we can simply add more virtual edges. For example, we can use $\mathbf{A} + \mathbf{A}^2$ as adjacency matrix. Another option is to add a virtual node that connect to all nodes to the graph.

4.3.5 Regularization on Graphs

Many of the standard regularization techniques can be applied on graphs, for example: L2 regularization, dropout. Moreover, there are regularization strategies that are specific to GNNs.

Parameter Sharing Across Layers: For a network that have many layers, specific set of parameters across layers can be shared. For example, (Li et al., 2015) uses the same set of weight matrices in Gated Recurrent Units (GRUs) of different layers. The reason is that the number of parameters grows linearly with the number of layers, thus make training difficult for massive graphs. Moreover, this parameter sharing technique is also widely used in multi-relational GNNs. It is because the number of parameters in these GNNs also scales linearly with the number of relation types, which further overblows the number of learnable parameters [9]. However, when using this strategy, one needs to be extremely careful of the risk of vanishing or exploding gradients [15]. By looking at the end-to-end formula for a GCN:

$$\mathbf{H}^{(L)} = \breve{\mathbf{A}}\sigma\Big(\breve{\mathbf{A}}\sigma\big(\ldots\sigma(\breve{\mathbf{A}}\mathbf{X}\mathbf{W}^{(0)})\ldots\big)\mathbf{W}^{(L-2)}\Big)\mathbf{W}^{(L-1)},$$

we can see that in case $\mathbf{W} = \mathbf{W}^{(0)} = \mathbf{W}^{(1)} = \ldots = \mathbf{W}^{(L)}$, the scale of our output will be blown up or shrunken by an approximated factor of $||\mathbf{W}||^L$.

Edge Dropout: The idea is to randomly mask edges during training, with the hope that this will make the network more robust to noise in the adjacency matrix. The neighborhood sampling technique used by GAT can also be considered as a special case of edge dropout. For example, given a graph G, DropEdge (Yu., 2020) first computes a new adjacency matrix $\mathbf{A}_d$ for each layer,

$$\mathbf{A}_d^{(\ell)} = \mathbf{A} - \mathbf{A}'^{(\ell)}, \tag{4.15}$$

where $\mathbf{A}'^{(\ell)}$ is a matrix obtained from a subset of edges of G. Then renormalization tricks are applied to obtain $\tilde{\mathbf{A}}$:

$$\tilde{\mathbf{A}}_{rd}^{(\ell)} = \widehat{\mathbf{D}}^{(\ell)^{-\frac{1}{2}}}(\mathbf{A}_d^{(\ell)} + \mathbf{I})\widehat{\mathbf{D}}^{(\ell)^{-\frac{1}{2}}}, \tag{4.16}$$

where $\widehat{\mathbf{D}}^{(\ell)}$ is the degree matrix associated with $(\mathbf{A}_d^{(\ell)} + \mathbf{I})$. Then a single-layer formula of GCN with edge dropout is given by

$$\mathbf{H}^{(\ell+1)} = \sigma(\tilde{\mathbf{A}}_{rd}^{(\ell+1)}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}). \tag{4.17}$$

Chapter 5

Numerical Experiments

In this chapter, we describes several implementations of PinSAGE architecture for a user-product recommendation problem.

5.1 Dataset and Task

The experiments are conducted on MovieLens-1M dataset, a dataset contains 1 million ratings from 6000 users on 4000 movies [2].

5.1.1 Data Format

Information about users, movies, and ratings are provided in 3 files: **users.dat**, **movies.dat** and **ratings.dat**, respectively. User information is in the following format:

$$UserID :: Gender :: Age :: Occupation :: ZipCode \tag{5.1}$$

All demographic information is provided voluntarily by the users and is not checked for accuracy. Only users who have provided some demographic information are included in this data set. Movie information is in the following format:

$$MovieID :: Title :: Genre \tag{5.2}$$

Only movies with at least 20 ratings are included. Ratings information is in the following format:

$$UserID :: MovieID :: Rating :: Timestamp \tag{5.3}$$

Table 5.1: User Features

Feature	Description
UserID	ID of the user, integers in range [1, 6040]
Gender	Gender of the user ('M' for male and 'F' for female)
Age	Age of the user, in 1 of 6 groups (see Appendix)
ZipCode	Postal code of the user's location

Table 5.2: Movie Features

Feature	Description
MovieID	ID of the movie
Title	String of words, taken from IMDB
Genre	Genre (one or many, selected from 18 different genres) (see Appendix)

Table 5.3: Rating Features

Feature	Description
UserID	ID of the user
MovieID	ID of the movie
Rating	Integer in [1, 5]
Timestamp	Timestamp of the rating, number of seconds passed from epoch

Table 5.4: List of Python libraries used and their versions

Libraries/Packages	Version
Python	3.9.7
PyTorch (torch)	1.11.0+cu113
torchtext	0.5.0
numpy	1.20.3
pandas	1.3.4
scipy	1.7.1
dgl	0.6.1
dask	2021.10.0
tqdm	4.62.3

5.1.2 Task

Given a training set including a list of users, a list of movies, and the lists of movies each user has watched, we would like to predict the next movie a user will watch.

5.2 Environment

5.2.1 Libraries

The end-to-end model pipeline, including preprocessing data, building models, training models, and performance evaluation are written in Python. The full list of Python libraries we used to produce experimental results is listed in the table below. Among them, PyTorch is a Python-based scientific computing package serving as a replacement for NumPy to use the power of GPUs and other accelerators, and an automatic differentiation library that is useful to implement neural networks [3]. Deep Graph Library (DGL) is a Python package built for easy implementation of graph neural network model family [1], on top of PyTorch.

5.2.2 Experiment Environment

All the experiments are conducted using a machine with the following configurations:

Table 5.5: Machine configuration

Specification	Configuration
Processor	Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz, 2.496Mhz, 4 Cores, 4 Logical Processors
Memory	8GB
System type	64-bit operating system, x64-based processor
GPU	NVIDIA GeForce GTX 1050
Operation System	Windows 10 Pro, build 18363.1556

To accelerate the training, we utilized the power of GPU for parallel computing. In order to do that, CUDA Toolkit (version 11.6) was downloaded and installed from NVIDIA's website.

5.3 General Approach

A bipartite graph $G = (V, E)$ is constructed from the dataset, where $V = V_U \cup V_M$, V_U is the set of nodes representing users, V_E is the set of nodes representing movies; and an edge (u, v) between user $u \in V_U$ and item $v \in V_E$ means that user u watches movie v.

5.3.1 Dataset Split

Dataset is split into 3 disjoint subsets: train, validation, and test. For each user $u \in V_U$, let $M(u) = \{v_1, \ldots, v_{n_u}\}$ where $n_u = |M(u)|$ be the list of movies that user u watches, sorted by ascending order of timestamp. For any u such that $|M(u)| \geqslant 2$, the training portion corresponding to user u consists of all the edges from u, except the last two:

$$E_u^{train} = \{(u, v_1), \ldots, (u, v_{n_u-2})\} \tag{5.4}$$

The validation portion corresponding to user u consists of the second last movie watched by user u:

$$E_u^{val} = \{(u, v_{n_u-1})\}, \tag{5.5}$$

and the test portion corresponding to user u consists of the last movie watched by user u:

$$E_u^{test} = \{(u, v_{n_u})\}. \tag{5.6}$$

The training graph is constructed as $G_T = (V_T, E_T)$ where $V_T = V$ and $E_T = \bigcup_{u \in V_U} E_u^{train}$. For $u \in V_U$ such that $|M(u)| = 0$, we do not have validation and test portions. For $u \in V_U$ such that $|M(u)| = 1$, we do not have validation portion.

5.3.2 Preprocessing

We keep all the features from Users and Movies tables. Regarding data types, there are two types of features in the dataset: numeric features and text features. Numeric features are already ready for training. Text features (movie titles) are embeded into numeric vectors using a single-layer neural network.

For Ratings table, we omit the ratings. Timestamps are only used to split the dataset into train/val/test. That means our graph G is an unweighted graph.

5.3.3 Constructing Samplers

Designing a good neighborhood sampler is a crucial part for the experiments. PinSAGE uses importance-based neighborhood sampling to choose a fixed-sized neighborhood for a graph node. For a given target node u, it performs random walks starting from u, then counts the occurrences of the nodes appearing on each walk, then finally choose the top T nodes with the highest visit counts.

A neighborhood sampling procedure consists of N_w random walks with restarts, all start from the source node u. The length for each walk is fixed and denoted as N_l. A walker is also equipped a restart capability, with probabillity $p_r \in [0, 1]$. When a restart occurs, the walker is forced come back to the source node, no matter where it was currently standing. The idea of introducing p_r is to give more preference to the nodes which are closer to the source node.

5.3.4 Constructing a Training Batch

Since the input graph is to big, we train the model using minibatches. Each batch B consists of a positive graph $P_{\mathcal{B}} = (V_{\mathcal{B}}^P, E_{\mathcal{B}}^P)$ and a negative graph $N_{\mathcal{B}} = (V_{\mathcal{B}}^N, E_{\mathcal{B}}^N)$ where $V_{\mathcal{B}}^P, V_{\mathcal{B}}^N \subseteq V$ and $E_{\mathcal{B}}^P, E_{\mathcal{B}}^N \subseteq E$.

The graphs are constructed from N_B triplets, each triplet $tr = (h, p, t)$ where $h, p, t \in V$. These triplets are chosen as follows. First, N_B movies $M_H = \{m_1, .., m_{N_B}\} \subseteq V$ are

drawn randomly and without replacement. These movies can be referred as the heads of the triplets. Next, N_B movies $M_P = \{p_1, .., p_{N_B}\} \subseteq V$ are chosen randomly such that $d(m_i, p_i) = 2$ where $d(\cdot, \cdot)$ is the shortest distance between two nodes in the graph. In words, we are chosen p_i such that p_i is co-interacted with m_i by some users. These movies are referred as positive tails of the triplets. Finally, other N_B movies $M_N = \{n_1, .., n_{N_B}\} \subseteq V$ are chosen randomly such that $d(m_i, n_i) > 2$, or in words, n_i are chosen such that n_i is not co-interacted with m_i by any user. The movies are referred as the negative tails of the triplet.

The set of nodes of the positive graph $P_{\mathcal{B}}$ is the union of the heads and positive tails, $V_{\mathcal{B}}^P = M_H \cup M_P$. The corresponding set of edges is $E_{\mathcal{B}}^P = \{(u, v) | u \in M_H, v \in M_P\}$. Similarly, the set of nodes of the positive graph $N_{\mathcal{B}}$ is the union of the heads and positive tails, $V_{\mathcal{B}}^N = M_H \cup N_P$. The corresponding set of edges is $E_{\mathcal{B}}^N = \{(u, v) | u \in M_H, v \in M_N\}$.

We choose Adam as the training optimizer.

5.3.5 Scoring and Loss Functions

Let $u, v \in V_M$. The score of an item pair is calculated as:

$$s(u, v) = h_u^T h_v \in \mathbb{R} \tag{5.7}$$

where $h_u, h_v \in \mathbb{R}^{(L)}$ are the output are the output embedding vectors of node u and node v

The score of a graph $R = (V_R, E_R)$ (can be positive or negative) is calculated as follow:

$$s_G(R) = \sum_{(u,v) \in V_R} s(u, v) \in \mathbb{R} \tag{5.8}$$

The loss over a training batch is given by the difference of its negative graph's score and its positive graph's score:

$$L(\mathcal{B}) = s_G(N_{\mathcal{B}}) - s_G(P_{\mathcal{B}}) \in \mathbb{R} \tag{5.9}$$

5.3.6 Model Architecture

We use a PinSAGE architecture [22] of two layers, including one hidden layer and the output layer. The dimension of the hidden layer is 16.

5.4 Evaluation

For this task, we aim to predict the next movie a user will watch. For each user $u \in V_U$, let $l_u \in V_M$ be the last movie that user u watched. The model returns K movies that are nearest neighbors of l_u (in terms of Euclidean distance) and serve them as recommendations to user u.

5.4.1 Hit@K

For a given user $u \in V_U$, let $a_u \in V_M$ be the next movie that u will watch, and R_u be the set of recommended movie for u, where $|R_u| = K$. The score of our model corresponding to user u is given as:

$$s_U(u) = \begin{cases} 1, & a_u \in R_u \\ 0, & \text{otherwise.} \end{cases}$$

The Hit@K (hit at K) metric of the model is calculated as:

$$Hit@K = \sum_{u \in V_U} s_U(u), \tag{5.10}$$

or in words, the number of users whose relevant items are in the recommended list of size K.

5.4.2 Evaluation Settings

After each epoch, the model is evaluated against the validation set and the test set. $Hit@K$ values are collected and charted.

5.5 Results and Extensions

5.5.1 Baseline Model

Following the PinSAGE architecture outlined in [22], a PinSAGE-based recommendation model was implemented and evaluated. The hyperparameters which are used for the model can be found in Table 5.6. The evaluation results are reported in Figure 5.1, 5.2, 5.3.

In the original PinSAGE model, some hyperparameters are fixed throughout the training phase, including restart probability, and number of random walks. It can be argued that there is no perfect hyperparameter value for all minibatch graphs: the optimal values are different and should depend on the size, or density of the graph.

In next sections, we propose two possible improvements to the original model. The ideas are very simple, they are just there to test the hypotheses that we propose. These improvements are deterministic and involve no randomness. Furthermore, for all the experiments, we keep the random seed of all the libraries/packages the same and equal to 1, so the result can easily be reproducible if necessary. More details can be found in Table 5.9.

5.5.2 Adaptive Restart Probability

Hypothesis: The optimal choice for restart probability of the random walks depends on the density of the graph.

In this experiment, we make the random walk restart probability depends on the density of the graph. Let p_0 be the initial probability value, $|E_m|$ be the number of edges of the minibatch graph, and $|E|$ be the number of edges in the original graph, then the new restart probability for random walks performing on that graph is calculated as:

$$p = p_0 - \frac{|E_m|}{|E|} \tag{5.11}$$

Under our training settings, with different hyperparameters, $|E_m|$ can range anywhere from 1,000 to 100,000. For the MovieLens dataset, $|E| \approx 1,000,000$. The set of hyperparameters ued for this model is reported in Table 5.7. The result of our experiment is shown in Figure 5.4.

5.5.3 Variable Number of Random Walks

Hypothesis: The optimal choice for number of performed random walks depends on the density of the graph.

In this experiment, we make the number of random walks depends on the density of the graph. The idea is that a more dense graph needs more random walks to identify

appropriate neighborhoods for the nodes. Let $|E_m|$ be the number of edges of the minibatch graph. Then the number of random walks that we perform is given by:

$$N_r = \begin{cases} 10 & \text{if } |E_m| < 50,000 \\ 30 & \text{otherwise.} \end{cases}$$

We compare this new model with the original model where N_r is fixed at 20. The results are shown in Figure 5.5, and reported in Table xx. The set of hyperparameters ued for this model is reported in Table 5.8. The new model does not quite outperform the original one. It is expected since once we perform a reasonably enough number of random walks, it does not matter whether do it several more or less times.

5.6 Observations

We recall that the training set consists of all but last 2 movies (ordered by timestamp) for each user. The validation set consists of the second last movie, and the test set consists of the last movie that each user watches. Therefore, it is expected that in most of the cases, a model's performance on validation set is better than test set. Our experiments confirms that empirically, for different values of K. From these experiments, we can conclude that in a recommender system task, time sensitiveness of data is very important. Predicting the next movie is much more easier than predicting the movie after that.

It can also be seen from the numerical experiments that as K getting bigger, the Hit@K metric on the validation set and test set are getting much more closer. This suggests us that our model is **effective**: most of our correct predictions are on top of the recommendation lists.

It can be seen from Experiment that the ARP model outperforms the original model with a fixed restart probability, even though the modification made is very simple and does not come with additional parameters. This result suggests that the optimal choice restart probability of the random walk algorithm should depend on the density of the edges. A more dense graph should have lower restart probability so we can more effectively take advantages of information from all the neighboring nodes.

From the Experiment, we can see that the VNRW model is somewhat better than the original model with a fixed number of random walks, but the result is not so convincing. There are multiple times during training that the VNRW is behind, and the variance in performance gain/loss among training iterations is large. One possible explanation is that once we perform a reasonably enough number of random walks, it does not matter whether do it several more or less times.

Figure 5.1: Numerical result for Experiment 1 (Original PinSAGE), K = 5

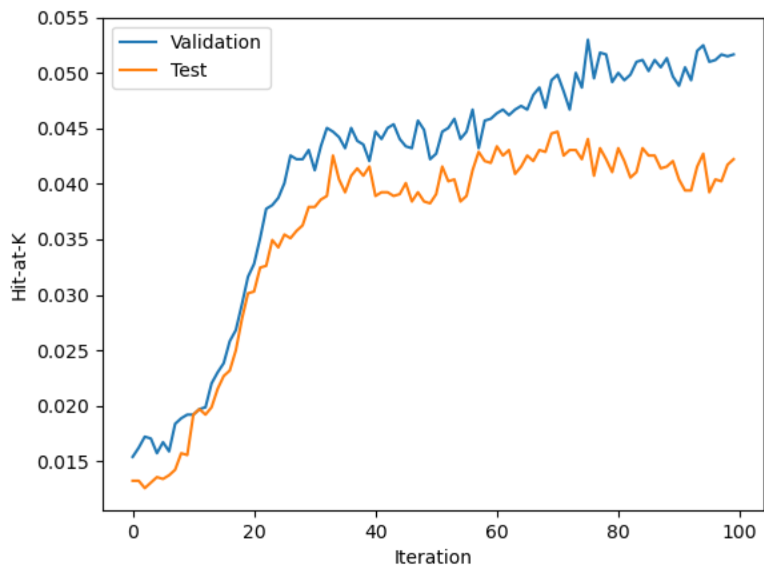


Figure 5.2: Numerical result for Experiment 2 (Original PinSAGE), K = 50

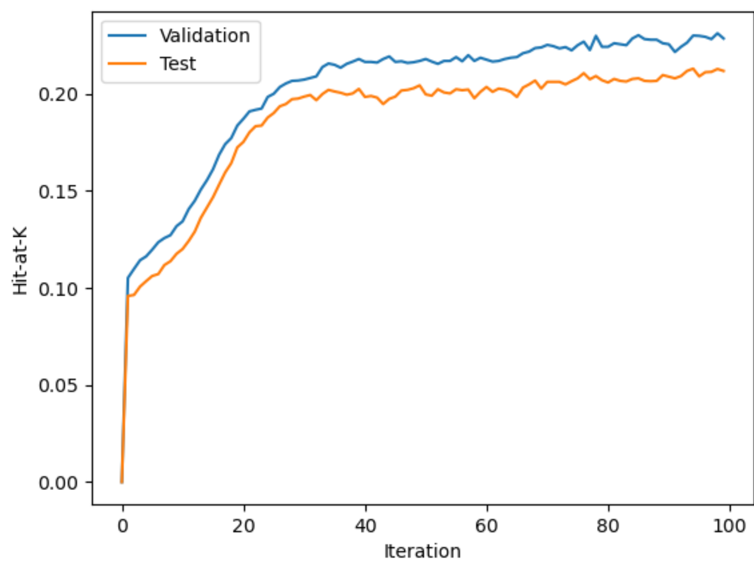


Figure 5.3: Numerical result for Experiment 3 (Original PinSAGE), K = 200

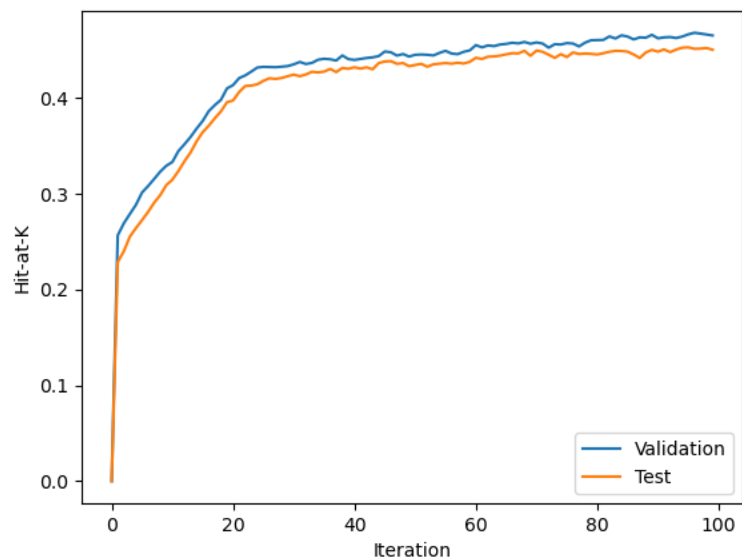


Figure 5.4: Experiment 4: Original PinSAGE vs Adaptive Restart Probability method

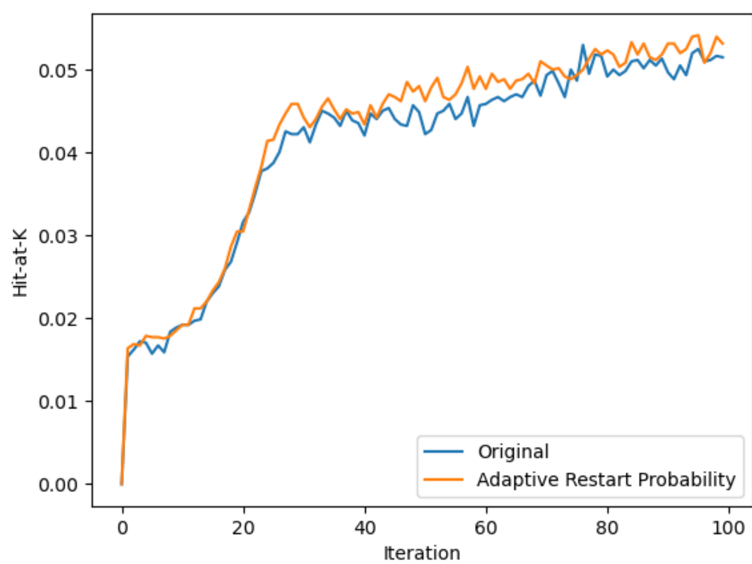


Figure 5.5: Experiment 5: Original PinSAGE vs Variable Number of Random Walks method

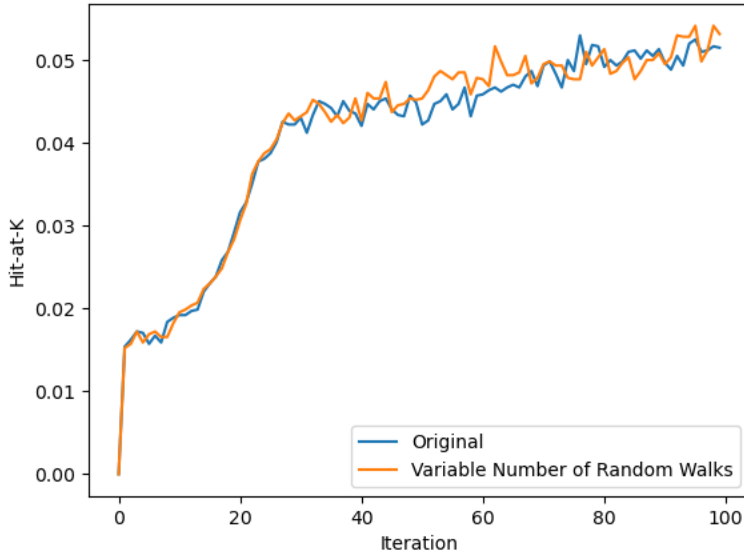


Table 5.6: Hyperparameters for the original PinSAGE (baseline)

Hyperparameter	Value
Random walk length	2
Random walk restart probability	0.5
Number of random walks	20
Neighborhood size	3
Number of layers	2
Hidden dimensions	16
Batch size	32
Number of epochs	100
Number of batches per epoch	1000
Learning rate	5E-5
K	5

Table 5.7: Hyperparameters for the Adaptive Restart Probability (ARP) method

Hyperparameter	**Value**
Random walk length	2
Number of random walks	20
Neighborhood size	3
Number of layers	2
Hidden dimensions	16
Batch size	32
Number of epochs	100
Number of batches per epoch	1000
Learning rate	5E-5
K	5

Table 5.8: Hyperparameters for the Variable Random Walk Lengths (VRWL) method

Hyperparameter	**Value**
Random walk length	2
Number of random walks (low)	10
Number of random walks (high)	30
Neighborhood size	3
Number of layers	2
Hidden dimensions	16
Batch size	32
Number of epochs	100
Number of batches per epoch	1000
Learning rate	5E-5
K	5

Table 5.9: Random seeds set for libraries/packages that involve randomness

Package/Library	Command
DGL	dgl.seed(1)
DGL Random	dgl.random.seed(1)
PyTorch	torch.manual_seed(1)
PyTorch CUDA	torch.cuda.manual_seed(1)
PyTorch CUDA (multiple GPUs)	torch.cuda.manual_seed_all(1)
PyTorch Backend	torch.backends.cudnn.deterministic=True
NumPy Random	np.random.seed(1)
Python Random	random.seed(1)

Chapter 6

Conclusions

6.1 Summary

The contribution of this thesis is two fold. First, we present a systematic way to write down popular GNN architectures, and the mathematical proof/foundation that is lacking in some original works. We also discuss the pros and cons and possible use cases of each architecture. Second, we implement a PinSAGE architecture to build a recommendation model for a user-product recommendation problem, propose additional improvements and run additional experiments to improve the result of the baseline model. Our experiements show that PinSAGE can be well adapted to a large-scale recommendation problem, and there are still some areas for improvements.

6.2 Future Work

It is already shown in [22] that PinSAGE can be used in web-scale recommendation system. However, in this work, due to limitation in computational power, we can only run the experiments on MovieLens-1M dataset. We believe that having an opportunity to run the experiments on bigger datasets, such as MovieLens-10M, MovieLens-25M, MovieLens-1B would open the door for more insights and widen the room for more improvement ideas.

In this work, I proposed hypotheses that the restart probability and number of random walks should be better adaptive to the sampled graph. I believe the same thing can be said for other hyperparameter of a random walk strategy. Planned future work will be

directed in this direction, aside from exploring other innovative ways to sample a node's neighborhood.

References

[1] Dgl.ai official website. https://docs.dgl.ai/. Accessed: 2022-05-01.

[2] Movielens 1m dataset. https://grouplens.org/datasets/movielens/1m/. Accessed: 2022-04-18.

[3] Pytorch official website. https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html. Accessed: 2022-05-01.

[4] X. Allan Chen. Understanding spectral graph neural network. 12 2020.

[5] Fan R. K. Chung. *Spectral Graph Theory*. 1997.

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems 29*, 8:175–191, 2016.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[9] William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

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

[11] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *ICLR 2017*.

[12] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. Personalized embedding propagation: Combining neural networks on graphs with personalized pagerank. *CoRR*, abs/1810.05997, 2018.

[13] Fei-Fei Li. Lecture notes in cs231n: Convolutional neural networks for visual recognition.

[14] Zhiyuan Liu and Jie Zhou. *Introduction to Graph Neural Networks. Synthesis Lectures on Artificial Intelligence and Machine Learning*. 2020.

[15] Denis Lukovnikov and Asja Fischer. Improving breadth-wise backpropagation in graph neural networks helps learning long-range dependencies. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 7180–7191. PMLR, 18–24 Jul 2021.

[16] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018.

[17] Luis G. Serrano. *Grokking Machine Learning*. Manning Publications, November 2021.

[18] David I Shuman, Sunil K. Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

[19] Nathan Srebro and Adi Shraibman. Rank, trace-norm and max-norm. In *Proceedings of the 18th Annual Conference on Learning Theory*, COLT'05, page 545–560, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] Petar Velivcković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. 2017.

[21] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 6861–6871. PMLR, 09–15 Jun 2019.

[22] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. *CoRR*, abs/1806.01973, 2018.

[23] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *CoRR*, abs/1806.08804, 2018.

[24] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.