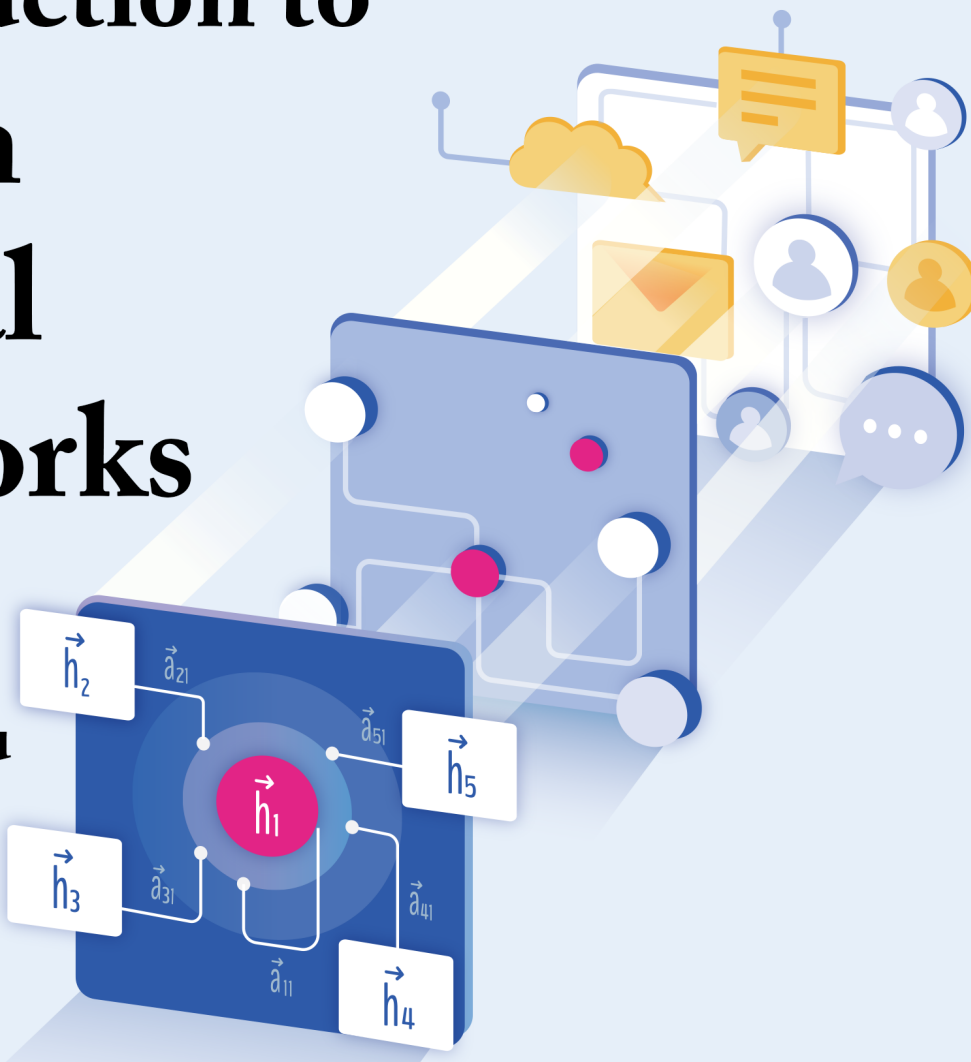
MORGAN & CLAYPOOL PUBLISHERS

Introduction to Graph Neural Networks



Zhiyuan Liu
Jie Zhou

Synthesis Lectures on Artificial Intelligence and Machine Learning

Ronald J. Brachman, Francesca Rossi, and Peter Stone, *Series Editors*

Introduction to Graph Neural Networks

Synthesis Lectures on Artificial Intelligence and Machine Learning

Judgment Aggregation: A Primer
Davide Grossi and Gabriella Pigozzi
2014

An Introduction to Constraint-Based Temporal Reasoning
Roman Barták, Robert A. Morris, and K. Brent Venable
2014

Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms
Rina Dechter
2013

Introduction to Intelligent Systems in Traffic and Transportation
Ana L.C. Bazzan and Franziska Klügl
2013

A Concise Introduction to Models and Methods for Automated Planning
Hector Geffner and Blai Bonet
2013

Essential Principles for Autonomous Robotics
Henry Hexmoor
2013

Case-Based Reasoning: A Concise Introduction
Beatriz López
2013

Answer Set Solving in Practice
Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub
2012

Planning with Markov Decision Processes: An AI Perspective
Mausam and Andrey Kolobov
2012

Active Learning
Burr Settles
2012

Computational Aspects of Cooperative Game Theory
Georgios Chalkiadakis, Edith Elkind, and Michael Wooldridge
2011

Representations and Techniques for 3D Object Recognition and Scene Interpretation
Derek Hoiem and Silvio Savarese
2011

A Short Introduction to Preferences: Between Artificial Intelligence and Social Choice
Francesca Rossi, Kristen Brent Venable, and Toby Walsh
2011

Human Computation
Edith Law and Luis von Ahn
2011

Trading Agents
Michael P. Wellman
2011

Visual Object Recognition
Kristen Grauman and Bastian Leibe
2011

Learning with Support Vector Machines
Colin Campbell and Yiming Ying
2011

Algorithms for Reinforcement Learning
Csaba Szepesvári
2010

Data Integration: The Relational Logic Approach
Michael Genesereth
2010

Markov Logic: An Interface Layer for Artificial Intelligence
Pedro Domingos and Daniel Lowd
2009

Introduction to Semi-Supervised Learning
Xiaojin Zhu and Andrew B.Goldberg
2009

Action Programming Languages
Michael Thielscher
2008

Representation Discovery using Harmonic Analysis
Sridhar Mahadevan
2008

Essentials of Game Theory: A Concise Multidisciplinary Introduction
Kevin Leyton-Brown and Yoav Shoham
2008

A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence
Nikos Vlassis
2007

Intelligent Autonomous Robotics: A Robot Soccer Case Study
Peter Stone
2007

Introduction to Graph Neural Networks

Zhiyuan Liu and Jie Zhou

www.morganclaypool.com

ISBN: 9781681737652 paperback
ISBN: 9781681737669 ebook
ISBN: 9781681737676 hardcover

DOI 10.2200/S00980ED1V01Y202001AIM045

A Publication in the Morgan & Claypool Publishers series
SYNTHESIS LECTURES ON ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING

Lecture #45
Series Editors: Ronald Brachman, *Jacobs Technion-Cornell Institute at Cornell Tech*
Francesca Rossi, *IBM Research AI*
Peter Stone, *University of Texas at Austin*
Series ISSN
Synthesis Lectures on Artificial Intelligence and Machine Learning
Print 1939-4608 Electronic 1939-4616

Introduction to Graph Neural Networks

Zhiyuan Liu and Jie Zhou
Tsinghua University

SYNTHESIS LECTURES ON ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING #45

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

Graphs are useful data structures in complex real-life applications such as modeling physical systems, learning molecular fingerprints, controlling traffic networks, and recommending friends in social networks. However, these tasks require dealing with non-Euclidean graph data that contains rich relational information between elements and cannot be well handled by traditional deep learning models (e.g., convolutional neural networks (CNNs) or recurrent neural networks (RNNs)). Nodes in graphs usually contain useful feature information that cannot be well addressed in most unsupervised representation learning methods (e.g., network embedding methods). Graph neural networks (GNNs) are proposed to combine the feature information and the graph structure to learn better representations on graphs via feature propagation and aggregation. Due to its convincing performance and high interpretability, GNN has recently become a widely applied graph analysis tool.

This book provides a comprehensive introduction to the basic concepts, models, and applications of graph neural networks. It starts with the introduction of the vanilla GNN model. Then several variants of the vanilla model are introduced such as graph convolutional networks, graph recurrent networks, graph attention networks, graph residual networks, and several general frameworks. Variants for different graph types and advanced training methods are also included. As for the applications of GNNs, the book categorizes them into structural, non-structural, and other scenarios, and then it introduces several typical models on solving these tasks. Finally, the closing chapters provide GNN open resources and the outlook of several future directions.

KEYWORDS

deep graph learning, deep learning, graph neural network, graph analysis, graph convolutional network, graph recurrent network, graph residual network

Contents

Preface

Deep learning has achieved promising progress in many fields such as computer vision and natural language processing. The data in these tasks are usually represented in the Euclidean domain. However, many learning tasks require dealing with non-Euclidean graph data that contains rich relational information between elements, such as modeling physical systems, learning molecular fingerprints, predicting protein interface, etc. Graph neural networks (GNNs) are deep learning-based methods that operate on graph domains. Due to its convincing performance and high interpretability, GNN has recently been a widely applied graph analysis method.

The book provides a comprehensive introduction to the basic concepts, models, and applications of graph neural networks. It starts with the basics of mathematics and neural networks. In the first chapters, it gives an introduction to the basic concepts of GNNs, which aims to provide a general overview for readers. Then it introduces different variants of GNNs: graph convolutional networks, graph recurrent networks, graph attention networks, graph residual networks, and several general frameworks. These variants tend to generalize different deep learning techniques into graphs, such as convolutional neural network, recurrent neural network, attention mechanism, and skip connections. Further, the book introduces different applications of GNNs in structural scenarios (physics, chemistry, knowledge graph), non-structural scenarios (image, text) and other scenarios (generative models, combinatorial optimization). Finally, the book lists relevant datasets, open source platforms, and implementations of GNNs.

This book is organized as follows. After an overview in Chapter 1, we introduce some basic knowledge of math and graph theory in Chapter 2. We show the basics of neural networks in Chapter 3 and then give a brief introduction to the vanilla GNN in Chapter 4. Four types of models are introduced in Chapters 5, 6, 7, and 8, respectively. Other variants for different graph types and advanced training methods are introduced in Chapters 9 and 10. Then we propose several general GNN frameworks in Chapter 11. Applications of GNN in structural scenarios, nonstructural scenarios, and other scenarios are presented in Chapters 12, 13, and 14. And finally, we provide some open resources in Chapter 15 and conclude the book in Chapter 16.

Zhiyuan Liu and Jie Zhou
March 2020

Acknowledgments

We would like to thank those who contributed and gave advice in individual chapters:

Chapter 1: Ganqu Cui, Zhengyan Zhang

Chapter 2: Yushi Bai

Chapter 3: Yushi Bai

Chapter 4: Zhengyan Zhang

Chapter 9: Zhengyan Zhang, Ganqu Cui, Shengding Hu

Chapter 10: Ganqu Cui

Chapter 12: Ganqu Cui

Chapter 13: Ganqu Cui, Zhengyan Zhang

Chapter 14: Ganqu Cui, Zhengyan Zhang

Chapter 15: Yushi Bai, Shengding Hu

We would also thank those who provide feedback on the content of the book: Cheng Yang, Ruidong Wu, Chang Shu, Yufeng Du, and Jiayou Zhang.

Finally, we would like to thank all the editors, reviewers, and staff who helped with the publication of the book. Without you, this book would not have been possible.

Zhiyuan Liu and Jie Zhou
March 2020

CHAPTER 1

Introduction

Graphs are a kind of data structure which models a set of objects (nodes) and their relationships (edges). Recently, researches of analyzing graphs with machine learning have received more and more attention because of the great expressive power of graphs, i.e., graphs can be used as denotation of a large number of systems across various areas including social science (social networks) [Hamilton et al., 2017b, Kipf and Welling, 2017], natural science (physical systems [Battaglia et al., 2016, Sanchez et al., 2018] and protein-protein interaction networks [Fout et al., 2017]), knowledge graphs [Hamaguchi et al., 2017] and many other research areas [Khalil et al., 2017]. As a unique non-Euclidean data structure for machine learning, graph draws attention on analyses that focus on node classification, link prediction, and clustering. Graph neural networks (GNNs) are deep learning-based methods that operate on graph domain. Due to its convincing performance and high interpretability, GNN has been a widely applied graph analysis method recently. In the following paragraphs, we will illustrate the fundamental motivations of GNNs.

1.1 MOTIVATIONS

1.1.1 CONVOLUTIONAL NEURAL NETWORKS

Firstly, GNNs are motivated by convolutional neural networks (CNNs) LeCun et al. [1998]. CNNs is capable of extracting and composing multi-scale localized spatial features for features of high representation power, which have result in breakthroughs in almost all machine learning areas and the revolution of deep learning. As we go deeper into CNNs and graphs, we find the keys of CNNs: local connection, shared weights, and the use of multi-layer [LeCun et al., 2015]. These are also of great importance in solving problems of graph domain, because (1) graphs are the most typical locally connected structure, (2) shared weights reduce the computational cost compared with traditional spectral graph theory [Chung and Graham, 1997], (3) multi-layer structure is the key to deal with hierarchical patterns, which captures the features of various sizes. However, CNNs can only operate on regular Euclidean data like images (2D grid) and text (1D sequence), which can also be regarded as instances of graphs. Therefore, it is straightforward to think of finding the generalization of CNNs to graphs. As shown in Figure 1.1, it is hard to define localized convolutional filters and pooling operators, which hinders the transformation of CNN from Euclidean to non-Euclidean domain.

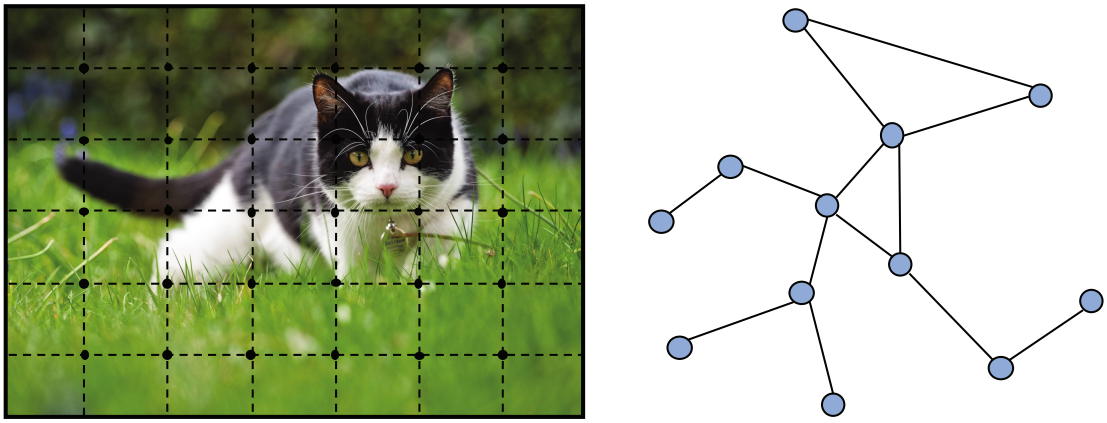


Figure 1.1: Left: image in Euclidean space. Right: graph in non-Euclidean space.

1.1.2 NETWORK EMBEDDING

The other motivation comes from *graph embedding* [Cai et al., 2018, Cui et al., 2018, Goyal and Ferrara, 2018, Hamilton et al., 2017a, Zhang et al., 2018a], which learns to represent graph nodes, edges, or subgraphs in low-dimensional vectors. In graph analysis, traditional machine learning approaches usually rely on hand-engineered features and are limited by its inflexibility and high cost. Following the idea of *representation learning* and the success of word embedding [Mikolov et al., 2013], DeepWalk [Perozzi et al., 2014], which is regarded as the first graph embedding method based on representation learning, applies SkipGram model [Mikolov et al., 2013] on the generated random walks. Similar approaches such as node2vec [Grover and Leskovec, 2016], LINE [Tang et al., 2015], and TADW [Yang et al., 2015b] also achieved breakthroughs. However, these methods suffer from two severe drawbacks [Hamilton et al., 2017a]. First, no parameters are shared between nodes in the encoder, which leads to computational inefficiency, since it means the number of parameters grows linearly with the number of nodes. Second, the direct embedding methods lack the ability of generalization, which means they cannot deal with dynamic graphs or be generalized to new graphs.

1.2 RELATED WORK

There exist several comprehensive reviews on GNNs. Monti et al. [2017] propose a unified framework, MoNet, to generalize CNN architectures to non-Euclidean domains (graphs and manifolds) and the framework could generalize several spectral methods on graphs [Atwood and Towsley, 2016, Kipf and Welling, 2017] as well as some models on manifolds [Boscaini et al., 2016, Masci et al., 2015]. Bronstein et al. [2017] provide a thorough review of geometric deep learning, which presents its problems, difficulties, solutions, applications, and future directions. Monti et al. [2017] and Bronstein et al. [2017] focus on generalizing convolutions to graphs

or manifolds, while in this book we only focus on problems defined on graphs and we also investigate other mechanisms used in GNNs such as gate mechanism, attention mechanism, and skip connections. Gilmer et al. [2017] propose the message passing neural network (MPNN) which could generalize several GNN and graph convolutional network approaches. Wang et al. [2018b] propose the non-local neural network (NLNN) which unifies several "self-attention"-style methods. However, in the original paper, the model is not explicitly defined on graphs. Focusing on specific application domains, Gilmer et al. [2017] and Wang et al. [2018b] only give examples of how to generalize other models using their framework and they do not provide a review over other GNN models. Lee et al. [2018b] provides a review over graph attention models. Battaglia et al. [2018] propose the graph network (GN) framework which has a strong capability to generalize other models. However, the graph network model is highly abstract and Battaglia et al. [2018] only give a rough classification of the applications.

Zhang et al. [2018f] and Wu et al. [2019c] are two comprehensive survey papers on GNNs and they mainly focus on models of GNN. Wu et al. [2019c] categorize GNNs into four groups: recurrent graph neural networks (RecGNNs), convolutional graph neural networks (ConvGNNs), graph auto-encoders (GAEs), and spatial-temporal graph neural networks (STGNNs). Our book has a different taxonomy with Wu et al. [2019c]. We present graph recurrent networks in Chapter 6. Graph convolutional networks are introduced in Chapter 5 and a special variant of ConvGNNs, graph attention networks, are introduced in Chapter 7. We present the graph spatial-temporal networks in Section 9.4 as the models are usually used on dynamic graphs. We introduce graph auto-encoders in Section 10.4 as they are trained in an unsupervised fashion.

In this book, we provide a thorough introduction to different GNN models as well as a systematic taxonomy of the applications. To summarize, the major contents of this book are as follows.

- We provide a detailed review over existing GNN models. We introduce the original model, its variants and several general frameworks. We examine various models in this area and provide a unified representation to present different propagation steps in different models. One can easily make a distinction between different models using our representation by recognizing corresponding aggregators and updaters.
- We systematically categorize the applications and divide them into structural scenarios, non-structural scenarios, and other scenarios. For each scenario, we present several major applications and their corresponding methods.

CHAPTER 2

Basics of Math and Graph

2.1 LINEAR ALGEBRA

The language and concepts of linear algebra have come into widespread usage in many fields in computer science, and machine learning is no exception. A good comprehension of machine learning is based upon a thoroughly understanding of linear algebra. In this section, we will briefly review some important concepts and calculations in linear algebra, which are necessary for understanding the rest of the book. In this section, we review some basic concepts and calculations in linear algebra which are necessary for understanding the rest of the book.

2.1.1 BASIC CONCEPTS

- **Scalar**: A number.
- **Vector**: A column of ordered numbers, which can be expressed as the following:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}. \tag{2.1}$$

The **norm** of a vector measures its length. The L_p norm is defined as follows:

$$||\mathbf{x}||_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{\frac{1}{p}}. \tag{2.2}$$

The L_1 norm, L_2 norm and L_∞ norm are often used in machine learning.
The $\mathbf{L_1}$ **norm** can be simplified as

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i|. \tag{2.3}$$

In Euclidean space $\mathbb{R}^n$, the $\mathbf{L_2}$ **norm** is used to measure the length of vectors, where

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} x_i^2}. \tag{2.4}$$

The $\mathbf{L}_\infty$ **norm** is also called the max norm, as

$$||\mathbf{x}||_\infty = \max_i |x_i|. \tag{2.5}$$

With L_p norm, the **distance** of two vectors $\mathbf{x}_1, \mathbf{x}_2$ (where $\mathbf{x}_1$ and $\mathbf{x}_2$ are in the same linear space) can be defined as

$$\mathbf{D}_p(\mathbf{x}_1, \mathbf{x}_2) = ||\mathbf{x}_1 - \mathbf{x}_2||_p. \tag{2.6}$$

A set of vectors $\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m$ are **linearly independent** if and only if there does not exist a set of scalars $\lambda_1, \lambda_2, \cdots, \lambda_m$, which are not all 0, such that

$$\lambda_1\mathbf{x}_1 + \lambda_2\mathbf{x}_2 + \cdots + \lambda_m\mathbf{x}_m = \mathbf{0}. \tag{2.7}$$

- **Matrix**: A two-dimensional array, which can be expressed as the following:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}, \tag{2.8}$$

where $\mathbf{A} \in \mathbb{R}^{m\times n}$.

Given two matrices $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times p}$, the **matrix product** of $\mathbf{AB}$ can be denoted as $\mathbf{C} \in \mathbb{R}^{m\times p}$, where

$$\mathbf{C}_{ij} = \sum_{k=1}^{n} \mathbf{A}_{ik}\mathbf{B}_{kj}. \tag{2.9}$$

It can be proved that matrix product is associative but not necessarily commutative. In mathematical language,

$$(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC}) \tag{2.10}$$

holds for arbitrary matrices **A, B**, and **C** (presuming that the multiplication is legitimate). Yet

$$\mathbf{AB} = \mathbf{BA} \tag{2.11}$$

is not always true.

For each $n \times n$ square matrix **A**, its **determinant** (also denoted as $|\mathbf{A}|$) is defined as

$$\det(\mathbf{A}) = \sum_{k_1k_2\cdots k_n} (-1)^{\tau(k_1k_2\cdots k_n)} a_{1k_1}a_{2k_2}\cdots a_{nk_n}, \tag{2.12}$$

where $k_1k_2\cdots k_n$ is a permutation of $1, 2, \cdots, n$ and $\tau(k_1k_2\cdots k_n)$ is the **inversion number** of the permutation $k_1k_2\cdots k_n$, which is the number of **inverted sequence** in the permutation.

If matrix $\mathbf{A}$ is a square matrix, which means that $m = n$, the **inverse matrix** of $\mathbf{A}$ (denoted as $\mathbf{A}^{-1}$) satisfies that

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}, \tag{2.13}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. $\mathbf{A}^{-1}$ exists if and only if $|\mathbf{A}| \neq 0$.
The **transpose** of matrix $\mathbf{A}$ is represented as $\mathbf{A}^T$, where

$$\mathbf{A}_{ij}^T = \mathbf{A}_{ji}. \tag{2.14}$$

There is another frequently used product between matrices called **Hadamard product**. The Hadamard product of two matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{m \times n}$ is a matrix $\mathbf{C} \in \mathbb{R}^{m \times n}$, where

$$\mathbf{C}_{ij} = \mathbf{A}_{ij}\mathbf{B}_{ij}. \tag{2.15}$$

- **Tensor**: An array with arbitrary dimension. Most matrix operations can also be applied to tensors.

2.1.2 EIGENDECOMPOSITION

Let $\mathbf{A}$ be a matrix in $\mathbb{R}^{n \times n}$. A nonzero vector $\mathbf{v} \in \mathbb{C}^n$ is called an **eigenvector** of $\mathbf{A}$ if there exists such scalar $\lambda \in \mathbb{C}$ that

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{2.16}$$

Here scalar λ is an **eigenvalue** of $\mathbf{A}$ corresponding to the eigenvector $\mathbf{v}$. If matrix $\mathbf{A}$ has n eigenvectors $\{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n\}$ that are linearly independent, corresponding to the eigenvalue $\{\lambda_1, \lambda_2, \cdots, \lambda_n\}$, then it can be deduced that

$$\mathbf{A}\begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n \end{bmatrix} \begin{bmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{bmatrix}. \tag{2.17}$$

Let $\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \dots & \mathbf{v}_n \end{bmatrix}$; then it is clear that $\mathbf{V}$ is an invertible matrix. We have the **eigendecomposition** of $\mathbf{A}$ (also called **diagonalization**)

$$\mathbf{A} = \mathbf{V}diag(\lambda)\mathbf{V}^{-1}. \tag{2.18}$$

It can also be written in the following form:

$$\mathbf{A} = \sum_{i=1}^{n} \lambda_i \mathbf{v}_i \mathbf{v}_i^T. \tag{2.19}$$

However, not all square matrices can be diagonalized in such form because a matrix may not have as many as n linear independent eigenvectors. Fortunately, it can be proved that every real symmetric matrix has an eigendecomposition.

2.1.3 SINGULAR VALUE DECOMPOSITION

As eigendecomposition can only be applied to certain matrices, we introduce the singular value decomposition, which is a generalization to all matrices.

First we need to introduce the concept of **singular value**. Let r denote the rank of $\mathbf{A}^T\mathbf{A}$, then there exist r positive scalars $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$ such that for $1 \leq i \leq r$, $\mathbf{v}_i$ is an eigenvector of $\mathbf{A}^T\mathbf{A}$ with corresponding eigenvalue σ_i^2. Note that $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_r$ are linearly independent. The r positive scalars $\sigma_1, \sigma_2, \cdots, \sigma_r$ are called singular values of $\mathbf{A}$. Then we have the singular value decomposition

$$\mathbf{A} = U\Sigma V^T, \tag{2.20}$$

where $U \in \mathbb{R}^{m\times m}$ and V $(n \times n)$ are orthogonal matrices and Σ is an $m \times n$ matrix defined as follows:

$$\Sigma_{ij} = \begin{cases} \sigma_i & \text{if } i = j \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

In fact, the column vectors of $\mathbf{U}$ are eigenvectors of $\mathbf{A}\mathbf{A}^T$, and the eigenvectors of $\mathbf{A}^T\mathbf{A}$ are made up of the the column vectors of $\mathbf{V}$.

2.2 PROBABILITY THEORY

Uncertainty is ubiquitous in the field of machine learning, thus we need to use probability theory to quantify and manipulate the uncertainty. In this section, we review some basic concepts and classic distributions in probability theory which are essential for understanding the rest of the book.

2.2.1 BASIC CONCEPTS AND FORMULAS

In probability theory, a **random variable** is a variable that has a random value. For instance, if we denote a random value by X, which has two possible values x_1 and x_2, then the probability of X equals to x_1 is $P(X = x_1)$. Clearly, the following equation remains true:

$$P(X = x_1) + P(X = x_2) = 1. \tag{2.21}$$

Suppose there is another random variable Y that has y_1 as a possible value. The probability that $X = x_1$ and $Y = y_1$ is written as $P(X = x_1, Y = y_1)$, which is called the **joint probability** of $X = x_1$ and $Y = y_1$.

Sometimes we need to know the relationship between random variables, like the probability of $X = x_1$ on the condition that $Y = y_1$, which can be written as $P(X = x_1|Y = y_1)$. We call this the **conditional probability** of $X = x_1$ given $Y = y_1$. With the concepts above, we can write the following two fundamental rules of probability theory:

$$P(X = x) = \sum_y P(X = x, Y = y), \tag{2.22}$$

$$P(X = x, Y = y) = P(Y = y|X = x)P(X = x). \tag{2.23}$$

The former is the **sum rule** while the latter is the **product rule**. Slightly modifying the form of product rule, we get another useful formula:

$$\begin{aligned} P(Y = y|X = x) &= \frac{P(X = x, Y = y)}{P(X = x)} \\ &= \frac{P(X = x|Y = y)P(Y = y)}{P(X = x)} \end{aligned} \tag{2.24}$$

which is the famous **Bayes formula**. Note that it also holds for more than two variables:

$$P(X_i = x_i|Y = y) = \frac{P(Y = y|X_i = x_i)P(X_i = x_i)}{\sum_{j=1}^{n} P(Y = y|X_j = x_j)P(X_j = x_j)}. \tag{2.25}$$

Using product rule, we can deduce the **chain rule**:

$$\begin{aligned} &P(X_1 = x_1, \cdots, X_n = x_n) \\ &= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i|X_1 = x_1, \cdots, X_{i-1} = x_{i-1}), \end{aligned} \tag{2.26}$$

where $X_1, X_2, \cdots, X_n$ are n random variables.

The average value of some function $f(x)$ (where x is the value of a certain random variable) under a probability distribution $P(x)$ is called the **expectation** of $f(x)$. For a discrete distribution, it can be written as

$$\mathbb{E}[f(x)] = \sum_{x} P(x)f(x). \tag{2.27}$$

Usually, when $f(x) = x$, $\mathbb{E}[x]$ stands for the expectation of x.

To measure the dispersion level of $f(x)$ around its mean value $\mathbb{E}[f(x)]$, we introduce the **variance** of $f(x)$:

$$\begin{aligned} Var(f(x)) &= \mathbb{E}[(f(x) - \mathbb{E}[f(x)])^2] \\ &= \mathbb{E}[f(x)^2] - \mathbb{E}[f(x)]^2. \end{aligned} \tag{2.28}$$

The **standard deviation** is the square root of variance. In some level, **covariance** expresses the degree to which two variables vary together:

$$Cov(f(x), g(y)) = \mathbb{E}[(f(x) - \mathbb{E}[f(x)])(g(y) - \mathbb{E}[g(y)])]. \tag{2.29}$$

Greater covariance indicates higher relevance between $f(x)$ and $g(y)$.

2.2.2 PROBABILITY DISTRIBUTIONS

Probability distributions describe the probability of a random variable or several random variables on every state. Several distributions that are useful in the area of machine learning are listed as follows.

- **Gaussian distribution**: it is also known as **normal distribution** and can be expressed as:

$$N\left(x|\mu,\sigma^2\right)=\sqrt{\frac{1}{2\pi\sigma^2}}\exp\left(-\frac{1}{2\sigma^2}(x-\mu)^2\right), \tag{2.30}$$

  where μ is the mean of variable x and σ^2 is the variance.

- **Bernoulli distribution**: random variable X can either be 0 or 1, with a probability $P(X=1)=p$. Then the distribution function is

$$P(X=x)=p^x(1-p)^{1-x}, x\in\{0,1\}. \tag{2.31}$$

  It is quite obvious that $E(X)=p$ and $Var(X)=p(1-p)$.

- **Binomial distribution**: repeat the Bernoulli experiment for N times and the times that X equals to 1 is denote by Y, then

$$P(Y=k)=\binom{N}{k}p^k(1-p)^{N-k} \tag{2.32}$$

  is the Binomial distribution satisfying that $E(Y)=np$ and $Var(Y)=np(1-p)$.

- **Laplace distribution**: the Laplace distribution is described as

$$P(x|\mu,b)=\frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right). \tag{2.33}$$

2.3 GRAPH THEORY

Graphs are the basic subjects in the study of GNNs. Therefore, to get a comprehensive understanding of GNN, the fundamental graph theory is required.

2.3.1 BASIC CONCEPTS

A graph is often denoted by $G=(V,E)$, where V is the set of **vertices** and E is the set of **edges**. An edge $e=u,v$ has two **endpoints** u and v, which are said to be **joined** by e. In this case, u is called a **neighbor** of v, or in other words, these two vertices are **adjacent**. Note that an edge can either be **directed** or **undirected**. A graph is called a **directed graph** if all edges are directed or **undirected graph** if all edges are undirected. The **degree** of vertice v, denoted by $d(v)$, is the number of edges connected with v.

2.3.2 ALGEBRA REPRESENTATIONS OF GRAPHS

There are a few useful algebra representations for graphs, which are listed as follows.

- **Adjacency matrix**: for a simple graph $G = (V, E)$ with n-vertices, it can be described by an adjacency matrix $A \in \mathbb{R}^{n\times n}$, where

$$A_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

  It is obvious that such matrix is a symmetric matrix when G is an undirected graph.

- **Degree matrix**: for a graph $G = (V, E)$ with n-vertices, its degree matrix $D \in \mathbb{R}^{n\times n}$ is a diagonal matrix, where

$$D_{ii} = d(v_i).$$

- **Laplacian matrix**: for a simple graph $G = (V, E)$ with n-vertices, if we consider all edges in G to be undirected, then its Laplacian matrix $L \in \mathbb{R}^{n\times n}$ can be defined as

$$L = D - A.$$

  Thus, we have the elements:

$$L_{ij} = \begin{cases} d(v_i) & \text{if } i = j, \\ -1 & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

  Note that the graph is considered to be an undirected graph for the adjacency matrix.

- **Symmetric normalized Laplacian**: the symmetric normalized Laplacian is defined as:

$$\begin{aligned} L^{sym} &= D^{-\frac{1}{2}} L D^{-\frac{1}{2}} \\ &= I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}. \end{aligned}$$

  The elements are given by:

$$L_{ij}^{sym} = \begin{cases} 1 & \text{if } i = j \text{ and } d(v_i) \neq 0, \\ -\frac{1}{\sqrt{d(v_i)d(v_j)}} & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

- **Random walk normalized Laplacian**: it is defined as:

$$L^{rw} = D^{-1} L = I - D^{-1} A.$$

  The elements can be computed by:

$$L_{ij}^{rw} = \begin{cases} 1 & \text{if } i = j \text{ and } d(v_i) \neq 0, \\ -\frac{1}{d(v_i)} & \text{if } \{v_i, v_j\} \in E \text{ and } i \neq j, \\ 0 & \text{otherwise.} \end{cases}$$

- **Incidence matrix**: another common used matrix in representing a graph is incidence matrix. For a directed graph $G = (V, E)$ with n-vertices and m-edges, the corresponding incidence matrix is $M \in \mathbb{R}^{n\times m}$, where

$$M_{ij} = \begin{cases} 1 & \text{if } \exists k \ s.t \ e_j = \{v_i, v_k\}, \\ -1 & \text{if } \exists k \ s.t \ e_j = \{v_k, v_i\}, \\ 0 & \text{otherwise.} \end{cases}$$

For a undirected graph, the corresponding incidence matrix satisfies that

$$M_{ij} = \begin{cases} 1 & \text{if } \exists k \ s.t \ e_j = \{v_i, v_k\}, \\ 0 & \text{otherwise.} \end{cases}$$

CHAPTER 3

Basics of Neural Networks

Neural networks are one of the most important models in machine learning. The structure of artificial neural networks, which consists of numerous neurons with connections to each other, bears great resemblance to that of biological neural networks. A neural network learns in the following way: initiated with random weights or values, the connections between neurons updates its weights or values by the back propagation algorithm repeatedly till the model performs rather precisely. In the end, the knowledge that a neural network learned is stored in the connections in a digital manner. Most of the researches on neural network try to change the way it learns (with different algorithms or different structures), aiming to improve the generalization ability of the model.

3.1 NEURON

The basic units of neural networks are **neurons**, which can receive a series of inputs and return the corresponding output. A classic neuron is as shown in Figure 3.1. Where the neuron receives n inputs $x_1, x_2, \cdots, x_n$ with corresponding weights $w_1, w_2, \cdots, w_n$ and an offset b. Then the weighted summation $y = \sum_{i=1}^{n} w_i x_i + b$ passes through an activation function f and the neuron returns the output $z = f(y)$. Note that the output will be the input of the next neuron. The **activation function** is a kind of function that maps a real number to a number between 0 and 1 (with rare exceptions), which represents the activation of the neuron, where 0 indicates deactivated and 1 indicates fully activated. Several useful activation functions are shown as follows.

- **Sigmoid Function** (Figure 3.2):

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3.1}$$

- **Tanh Function** (Figure 3.3):

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{3.2}$$

- **ReLU (Rectified Linear Unit)** (Figure 3.4):

$$ReLU(x) = \begin{cases} 0 & x \leq 0, \\ x & x > 0. \end{cases} \tag{3.3}$$

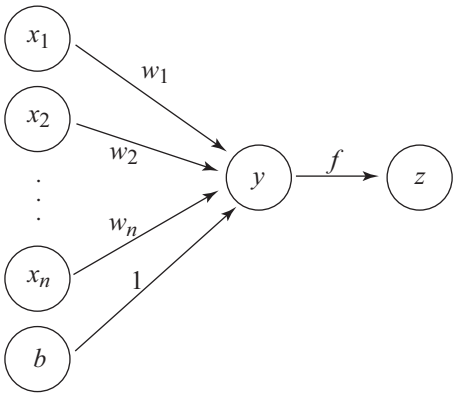


Figure 3.1: A classic neuron structure.

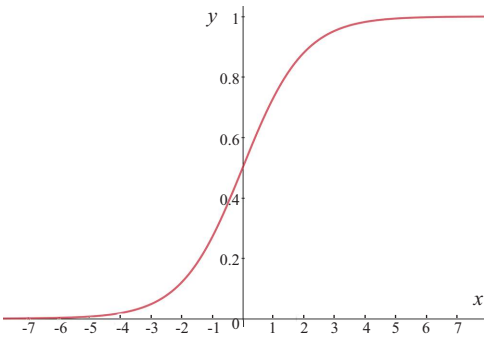


Figure 3.2: The Sigmoid function.

In fact, there are many other activation functions and each has its corresponding derivatives. But do remember that a good activation function is always smooth (which means that it is a continuous differentiable function) and easily calculated (in order to minimize the computational complexity of the neural network). During the training of a neural network, the choice of activation function is usually essential to the outcome.

3.2 BACK PROPAGATION

During the training of a neural network, the **back propagation algorithm** is most commonly used. It is an algorithm based on gradient descend to optimize the parameters in a model. Let's take the single neuron model illustrated above for an example. Suppose the optimization target for the output z is z_0, which will be approached by adjusting the parameters $w_1, w_2, \cdots, w_n, b$.

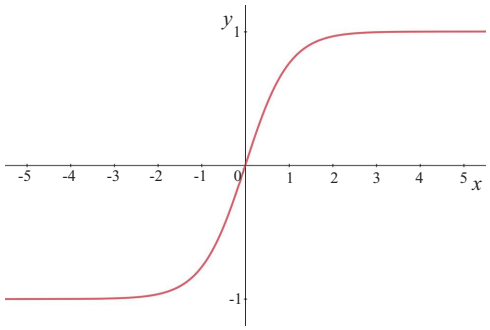


Figure 3.3: The Tanh function.

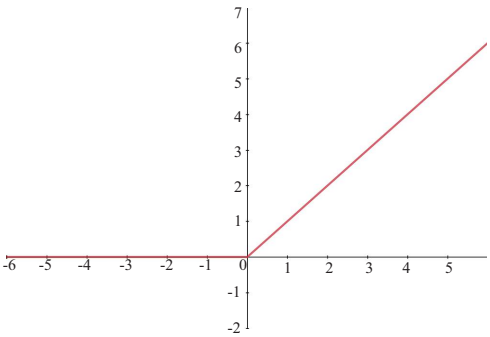


Figure 3.4: The ReLU (Rectified Linear Unit) function.

By the chain rule, we can deduce the derivative of z with respect to w_i and b:

$$\begin{aligned}\frac{\partial z}{\partial w_i} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial w_i} \\ &= \frac{\partial f(y)}{\partial y}x_i\end{aligned} \tag{3.4}$$

$$\begin{aligned}\frac{\partial z}{\partial b} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial b} \\ &= \frac{\partial f(y)}{\partial y}.\end{aligned} \tag{3.5}$$

With a learning rate of η, the update for each parameter will be:

$$\begin{aligned}\Delta w_i &= \eta(z_0 - z)\frac{\partial z}{\partial w_i} \\ &= \eta(z_0 - z)x_i\frac{\partial f(y)}{\partial y}\end{aligned} \tag{3.6}$$

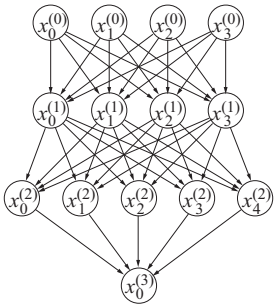


Figure 3.5: Feedforward neural network.

$$\begin{aligned}\Delta b &= \eta(z_0 - z)\frac{\partial z}{\partial b} \\ &= \eta(z_0 - z)\frac{\partial f(y)}{\partial y}.\end{aligned} \tag{3.7}$$

In summary, the process of the back propagation consists of the following two steps.

- **Forward calculation**: given a set of parameters and an input, the neural network computes the values at each neuron in a forward order.
- **Backward propagation**: compute the error at each variable to be optimized, and update the parameters with their corresponding partial derivatives in a backward order.

The above two steps will go on repeatedly until the optimization target is acquired.

3.3 NEURAL NETWORKS

Recently, there is a booming development in the field of machine learning (especially deep learning), represented by the appearance of a variety of neural network structures. Though varying widely, the current neural network structures can be classified into several categories: feedforward neural networks, convolutional neural networks, recurrent neural networks, and GNNs.

- **Feedforward neural network**: The feedforward neural network (FNN) (Figure 3.5) is the first and simplest network architecture of artificial neural network. The FNN usually contains an input layer, several hidden layers, and an output layer. The feedforward neural network has a clear hierarchical structure, which always consists of multiple layers of neurons, and each layer is only connected to its neighbor layers. There are no loops in this network.
- **Convolutional neural network**: Convolutional neural networks (CNNs) are special versions of FNNs. FNNs are usually fully connected networks while CNNs preserve the local

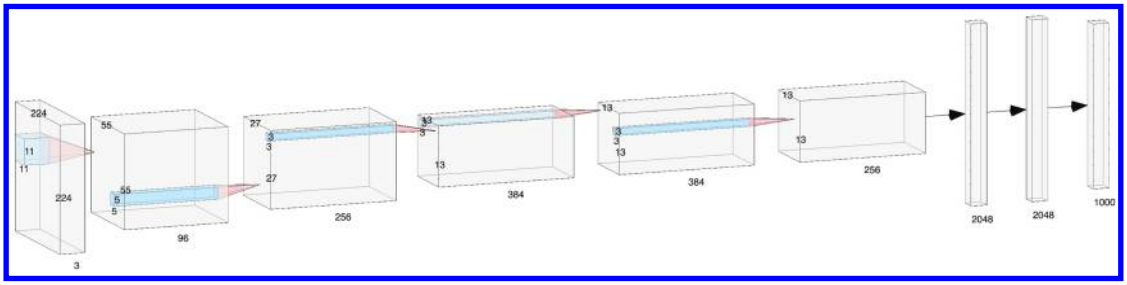


Figure 3.6: The AlexNet architecture from Krizhevsky et al. [2012].

connectivity. The CNN architecture usually contains convolutional layers, pooling layers, and several fully connected layers. There exist several classical CNN architectures such as LeNet5 [LeCun et al., 1998], AlexNet [Krizhevsky et al., 2012] (Figure 3.6), VGG [Simonyan and Zisserman, 2014], and GoogLeNet [Szegedy et al., 2015]. CNNs are widely used in the area of computer vision and proven to be effective in many other research fields.

- **Recurrent neural network**: In comparison with FNN, the neurons in recurrent neural network (RNN) receive not only signals and inputs from other neurons, but also its own historical information. The memory mechanism in recurrent neural network (RNN) help the model to process series data effectively. However, the RNN usually suffers from the problem of long-term dependencies [Bengio et al., 1994, Hochreiter et al., 2001]. Several variants are proposed to solve the problem by incorporating the gate mechanism such as GRU [Cho et al., 2014] and LSTM [Hochreiter and Schmidhuber, 1997]. The RNN is widely used in the area of speech and natural language processing.
- **Graph neural network**: The GNN is designed specifically to handle graph-structured data, such as social networks, molecular structures, knowledge graphs, etc. Detailed descriptions of GNNs will be covered in the later chapters of this book.

CHAPTER 4

Vanilla Graph Neural Networks

In this section, we describe the vanilla GNNs proposed in Scarselli et al. [2009]. We also list the limitations of the vanilla GNN in representation capability and training efficiency. After this chapter we will talk about several variants of the vanilla GNN model.

4.1 INTRODUCTION

The concept of GNN was first proposed in Gori et al. [2005], Scarselli et al. [2004, 2009]. For simplicity, we will talk about the model proposed in Scarselli et al. [2009], which aims to extend existing neural networks for processing graph-structured data.

A node is naturally defined by its features and related nodes in the graph. The target of GNN is to learn a state embedding $\mathbf{h}_v \in \mathbb{R}^s$, which encodes the information of the neighborhood, for each node. The state embedding $\mathbf{h}_v$ is used to produce an output $\mathbf{o}_v$, such as the distribution of the predicted node label.

In Scarselli et al. [2009], a typical graph is illustrated in Figure 4.1. The vanilla GNN model deals with the undirected homogeneous graph where each node in the graph has its input features $\mathbf{x}_v$ and each edge may also have its features. The paper uses $co[v]$ and $ne[v]$ to denote the set of edges and neighbors of node v. For processing other more complicated graphs such as heterogeneous graphs, the corresponding variants of GNNs could be found in later chapters.

4.2 MODEL

Given the input features of nodes and edges, next we will talk about how the model obtains the node embedding $\mathbf{h}_v$ and the output embedding $\mathbf{o}_v$.

In order to update the node state according to the input neighborhood, there is a parametric function f, called *local transition function*, shared among all nodes. In order to produce the output of the node, there is a parametric function g, called *local output function*. Then, $\mathbf{h}_v$ and $\mathbf{o}_v$ are defined as follows:

$$\mathbf{h}_v = f(\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}) \tag{4.1}$$

$$\mathbf{o}_v = g(\mathbf{h}_v, \mathbf{x}_v), \tag{4.2}$$

where $\mathbf{x}$ denotes the input feature and $\mathbf{h}$ denotes the hidden state. $co[v]$ is the set of edges connected to node v and $ne[v]$ is set of neighbors of node v. So that $\mathbf{x}_v, \mathbf{x}_{co[v]}, \mathbf{h}_{ne[v]}, \mathbf{x}_{ne[v]}$ are the features of v, the features of its edges, the states and the features of the nodes in the

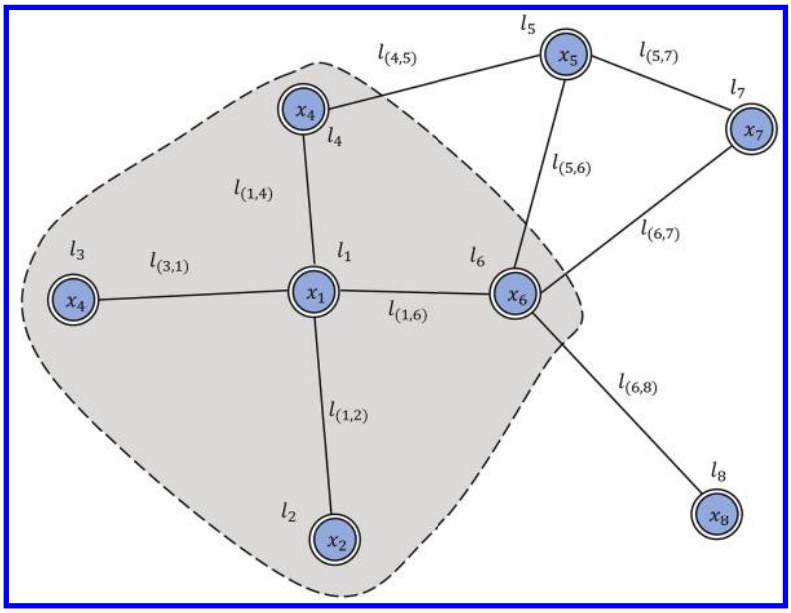


Figure 4.1: An example of the graph based on Scarselli et al. [2009].

neighborhood of v, respectively. In the example of node l_1 in Figure 4.1, $\mathbf{x}_{l_1}$ is the input feature of l_1. $co[l_1]$ contains edges $l_{(1,4)}, l_{(1,6)}, l_{(1,2)}$, and $l_{(3,1)}$. $ne[l_1]$ contains nodes l_2, l_3, l_4, and l_6.

Let $\mathbf{H}$, $\mathbf{O}$, $\mathbf{X}$, and $\mathbf{X}_N$ be the matrices constructed by stacking all the states, all the outputs, all the features, and all the node features, respectively. Then we have a compact form as:

$$\mathbf{H} = F(\mathbf{H}, \mathbf{X}) \tag{4.3}$$

$$\mathbf{O} = G(\mathbf{H}, \mathbf{X}_N) \tag{4.4}$$

where F, the *global transition function*, and G is the *global output function*. They are stacked versions of the local transition function f and the local output function g for all nodes in a graph, respectively. The value of $\mathbf{H}$ is the fixed point of Eq. (4.3) and is uniquely defined with the assumption that F is a contraction map.

With the suggestion of Banach's fixed point theorem [Khamsi and Kirk, 2011], GNN uses the following classic iterative scheme to compute the state:

$$\mathbf{H}^{t+1} = F(\mathbf{H}^t, \mathbf{X}), \tag{4.5}$$

where $\mathbf{H}^t$ denotes the tth iteration of $\mathbf{H}$. The dynamical system Eq. (4.5) converges exponentially fast to the solution of Eq. (4.3) for any initial value of $\mathbf{H}(0)$. Note that the computations described in f and g can be interpreted as the FNNs.

After the introduction of the framework of GNN, the next question is how to learn the parameters of the local transition function f and local output function g. With the target information ($\mathbf{t}_v$ for a specific node) for the supervision, the loss can be written as:

$$loss = \sum_{i=1}^{p} (\mathbf{t}_i - \mathbf{o}_i), \tag{4.6}$$

where p is the number of supervised nodes. The learning algorithm is based on a gradient-descent strategy and is composed of the following steps.

- The states $\mathbf{h}_v^t$ are iteratively updated by Eq. (4.1) until a time step T. Then we obtain an approximate fixed point solution of Eq. (4.3): $\mathbf{H}(T) \approx \mathbf{H}$.
- The gradient of weights $\mathbf{W}$ is computed from the loss.
- The weights $\mathbf{W}$ are updated according to the gradient computed in the last step.

After running the algorithm, we can get a model trained for a specific supervised/semi-supervised task as well as hidden states of nodes in the graph. The vanilla GNN model provides an effective way to model graphic data and it is the first step toward incorporating neural networks into graph domain.

4.3 LIMITATIONS

Though experimental results showed that GNN is a powerful architecture for modeling structural data, there are still several limitations of the vanilla GNN.

- First, it is computationally inefficient to update the hidden states of nodes iteratively to get the fixed point. The model needs T steps of computation to approximate the fixed point. If relaxing the assumption of the fixed point, we can design a multi-layer GNN to get a stable representation of the node and its neighborhood.
- Second, vanilla GNN uses the same parameters in the iteration while most popular neural networks use different parameters in different layers, which serves as a hierarchical feature extraction method. Moreover, the update of node hidden states is a sequential process which can benefit from the RNN kernels like GRU and LSTM.
- Third, there are also some informative features on the edges which cannot be effectively modeled in the vanilla GNN. For example, the edges in the knowledge graph have the type of relations and the message propagation through different edges should be different according to their types. Besides, how to learn the hidden states of edges is also an important problem.

- Last, if T is pretty large, it is unsuitable to use the fixed points if we focus on the representation of nodes instead of graphs because the distribution of representation in the fixed point will be much more smooth in value and less informative for distinguishing each node.

Beyond the vanilla GNN, several variants are proposed to release these limitations. For example, Gated Graph Neural Network (GGNN) [Li et al., 2016] is proposed to solve the first problem. Relational GCN (R-GCN) [Schlichtkrull et al., 2018] is proposed to deal with directed graphs. More details could be found in the following chapters.