‹packt›

1ST EDITION

Applied Deep Learning on Graphs

Leverage graph data for business applications using specialized deep learning architectures

LAKSHYA KHANDELWAL
SUBHAJOY DAS

Applied Deep Learning on Graphs

Leverage graph data for business applications using specialized deep learning architectures

Lakshya Khandelwal

Subhajoy Das

Applied Deep Learning on Graphs

Group Product Manager: Niranjan Naikwadi

Publishing Product Manager: Yasir Ali Khan

Book Project Manager: Aparna Nair

Senior Editor: Paridhi Agarwal

Technical Editor: Seemanjay Ameriya

Copy Editor: Safis Editing

Proofreader: Paridhi Agarwal

Indexer: Manju Arasan

Production Designers: Jyoti Kadam and Alishon Falcon

DevRel Marketing Coordinator: Vinishka Kalra

First published: December 2024

Production reference: 1121224

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-83588-596-3

`www.packtpub.com`

To my beloved parents, Ravi and Preeti, whose sacrifices helped me become who I am today. My deepest gratitude to my wife, Deblina, whose unwavering support and patience made those countless late nights of writing possible. Special thanks to my brother, Rahul, whose guidance has been my compass through the years, and Arushi, my sister-in-law, along with my precious eight-month-old niece, Nehal, whose smiles and laughter brought joy during this challenging journey. Your collective love, support, and encouragement made this book a reality.

– Lakshya Khandelwal

Contributors

About the authors

Lakshya Khandelwal holds a bachelor's and master's degree from IIT Kanpur in mathematics and computer science and has 8+ years of experience in building scalable machine learning products for multiple tech giants. He has worked as a lead ML engineer with Samsung, building natural language intelligence for the very first version of Bixby. He has also worked as a data scientist with Adobe, developing search bid optimization solutions as part of the advertising cloud suite for major enterprises across the globe. In addition, he has led natural language and forecasting initiatives at Walmart, building next-generation AI products for millions of customers. Lakshya currently leads AI for AirMDR, building agentic AI for the cybersecurity domain.

Subhajoy Das is a staff data scientist with 7 years of experience under his belt. He graduated from IIT Kharagpur with a bachelor's and master's degree in mathematics and computing. Since then, he has worked in organizations at varying stages of growth: from fast-growing e-commerce start-ups such as Meesho to behemoths such as Adobe. He has driven several pivotal features in every company he has worked in, including building an end-to-end recommendation system for the Meesho app and curating interesting advertising using reinforcement learning-based optimizations in Adobe Advertising. He is currently working at Arista Networks, building AI-driven apps that are responsible for the cybersecurity of several Fortune 500 companies.

About the reviewers

Sumit Dahiya is a seasoned cybersecurity specialist and solution architect with a focus on cloud security, identity and access management, and digital transformation. With more than 18 years of experience, he spearheads extensive security projects and digital transformation initiatives and is renowned for developing industry-leading solutions and leading multinational teams. He has experience with safe system architecture, microservices, and open-source technologies. Sumit mentors people in the fields of architecture and cybersecurity and has contributed to several papers and conferences. He wants to express his gratitude to his mentors, family, and friends for their constant encouragement and support along his journey.

Humashankar Vellathur Jaganathan is the principal engineering manager at CGI and a BCS Fellow. He is an esteemed mentor and key strategic adviser for Hubspot and Lucid Software; as a leader, he possesses a unique ability to strategize and think on his feet. His publications include "*Mortgage-based securities data hybrid encryption for financial data analysis*" in the *International Journal of Electronic Security and Digital Forensics*.

I would like to extend special thanks to my mentor, Prakash Murugesan, a distinguished engineer at Verizon, and Imran Ur Rehman, a senior project manager at Capgemini, for their expert guidance and valuable input.

Ashish Kumar is an AI and data science innovator with over 8 years of experience, specializing in scalable, real-time AI solutions. He holds an integrated MTech in mathematics and computing from IIT Delhi (2016). Ashish's groundbreaking work includes the development of a bidding algorithm for low-impression keywords within Adobe Advertising Cloud, for which he earned a U.S. patent. Recently, Ashish mastered large language models, successfully delivering a project for profile generation. He has served as a judge in Microsoft's hackathon, further demonstrating his expertise and leadership in AI. His work is marked by a proven ability to drive impactful, innovative solutions across complex, high-stakes applications.

Table of Contents

Part 3: Practical Applications and Implementation

Preface

In recent years, the rapid growth of networked data across fields such as social networks, molecular structures, recommendation systems, and computer vision has highlighted the need for better methods to process graph-structured data. Traditional deep learning models work well with grid-like data (such as images) and sequential data (such as text), but these models struggle with irregular structures such as graphs.

Welcome to *Applied Deep Learning on Graphs*, a book that addresses this challenge by exploring cutting-edge techniques at the intersection of graph theory and deep learning, offering practical strategies and insights from real-world applications at leading companies.

Drawing on our experiences implementing graph-based solutions at organizations such as Adobe, Walmart, and Meesho, the authors have seen firsthand how these techniques can transform business applications. This book provides a clear, comprehensive guide to understanding and applying graph deep learning, combining theoretical foundations with hands-on insights from large-scale industry implementations.

Who this book is for

This book serves multiple technical audiences. Data scientists, machine learning practitioners, and researchers looking to expand their expertise into graph-based data analysis will find practical guidance supported by real-world examples. Additionally, software engineers working on graph-related applications can benefit from the hands-on approach and implementation details provided here.

The content is equally valuable for tech leaders and entrepreneurs who need to understand the strategic implications of graph deep learning. We offer a comprehensive overview of how graph-based approaches can be applied across various domains, enabling decision-makers to identify opportunities and build solutions tailored to their specific business requirements. The book bridges the gap between theoretical concepts and practical implementation, making it a versatile reference for both technical execution and strategic planning.

What this book covers

Chapter 1, *Introduction to Graph Learning*, introduces graph learning, explaining how graphs can represent complex relationships more efficiently than traditional tabular data structures, demonstrated through various types of graphs, their properties, and computational representations.

Chapter 2, *Graph Learning in the Real World*, provides a comprehensive exploration of graph learning across three fundamental levels: the node level (predicting attributes of individual nodes), the edge level (analyzing relationships between nodes), and the graph level (studying entire graph structures), with practical applications in recommendation systems, knowledge graphs, cybersecurity, and **natural language processing** (**NLP**).

Chapter 3, *Graph Representation Learning*, explains graph representation learning techniques, focusing on shallow encoding methods such as DeepWalk and Node2Vec, which use random walks to generate node embeddings in graphs.

Chapter 4, *Deep Learning Models for Graphs*, provides a detailed exploration of **graph neural networks** (**GNNs**), covering message-passing mechanisms and three key architectures: **graph convolutional networks** (**GCNs**), GraphSAGE, and **graph attention networks** (**GATs**).

Chapter 5, *Graph Deep Learning Challenges*, exhaustively covers the major challenges in graph deep learning, including data-related issues, model architecture limitations, computational constraints, task-specific problems, and interpretability concerns in GNNs.

Chapter 6, *Harnessing Large Language Models for Graph Learning*, explores how **large language models** (**LLMs**) can enhance graph learning tasks through feature enhancement, prediction capabilities, and **retrieval-augmented generation** (**RAG**) approaches.

Chapter 7, *Graph Deep Learning in Practice*, demonstrates practical implementations of graph deep learning techniques for social network analysis using PyTorch Geometric, focusing on node classification and link prediction tasks in a university student network.

Chapter 8, *Graph Deep Learning for Natural Language Processing*, explores how graph deep learning techniques are applied to NLP tasks, covering linguistic graph structures, text summarization, information extraction, and dialogue systems.

Chapter 9, *Building Recommendation Systems Using Graph Deep Learning*, comprehensively covers how to build and implement recommendation systems using graph deep learning, including fundamental concepts, graph structures, model architectures, and training techniques.

Chapter 10, *Graph Deep Learning for Computer Vision*, covers how GNNs can enhance computer vision tasks by representing visual data as graphs, enabling better modeling of relationships and structural properties compared to traditional grid-based **convolutional neural network** (**CNN**) approaches.

Chapter 11, *Emerging Applications*, explores the latest practical applications of graph deep learning across six major domains: biology/healthcare, social networks, financial services, cybersecurity, energy systems, and **Internet of Things** (**IoT**).

Chapter 12, *The Future of Graph Learning*, explores the future trajectory of graph learning, covering emerging trends, advanced architectures, **artificial intelligence** (**AI**) integration, and potential breakthroughs in areas such as quantum computing, **artificial general intelligence** (**AGI**), and metaverse applications.

To get the most out of this book

To make the most of this book, you should have a foundation in machine learning fundamentals, basic graph theory, and programming concepts. We assume you're familiar with Python programming and have a working understanding of neural networks and deep learning principles. Prior exposure to data structures and algorithms will be beneficial but is not mandatory.

Software/hardware covered in the book	**Operating system requirements**
Jupyter Lab/Google Colab	Windows, macOS, or Linux
Python	

To successfully work with the examples in this book, you'll need to set up your Python environment with several essential packages. Start by installing the core libraries:

- **PyTorch** for deep learning operations
- **PyTorch Geometric** for graph-based neural networks
- **NetworkX** for graph manipulation and analysis
- **NumPy** and **pandas** for data handling and numerical computations

These can be installed using `pip` or `conda` package managers.

For optimal performance, ensure your system has at least 8 GB of RAM and a relatively modern CPU. While a GPU isn't strictly necessary for running smaller examples, having one will significantly speed up the training process for larger models and more complex graph operations.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Applied-Deep-Learning-on-Graphs`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "We set the model to evaluation mode using `model.eval()`."

A block of code is set as follows:

```
model.eval()
_, pred = model(data.x, data.edge_index).max(dim=1)
correct = float(pred[data.test_mask].eq(
    data.y[data.test_mask]).sum().item())
accuracy = correct / data.test_mask.sum().item()
print(f'Accuracy: {accuracy:.4f}')
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import torch
import torch.nn.functional as F
from torch_geometric.datasets import Planetoid
from torch_geometric.nn import GCNConv

# Load the Cora dataset
dataset = Planetoid(root='data/Cora', name='Cora')
```

Any command-line input or output is written as follows:

```
pip install torch torch_geometric scikit-learn matplotlib networkx
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "In an e-commerce system, one possible use case is to predict the **lifetime value** (**LTV**) of customers based on their interactions with products and other features."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Applied Deep Learning on Graphs*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Part 1: Foundations of Graph Learning

In the first part of the book, you will get an overview of the fundamental concepts of graph learning, including basic definitions, real-world applications, and core representation techniques. You will learn about the essential building blocks needed to understand graph-based machine learning, practical use cases across industries, and various methods for representing graph data in machine learning contexts.

This part has the following chapters:

1
Introduction to Graph Learning

Graph data is a powerful and intuitive way of expressing information, and several practical scenarios can be better expressed using graph data than tabular approaches. Analyzing graph data has been a topic of study for decades, but it has only recently begun to capture the limelight due to advances in compute capabilities.

In this book, we aim to introduce you to the world of **graphs**. Here, we'll begin by discussing what graph data is and the fundamental mathematical terminologies surrounding graphs. Next, we'll take a small detour and discuss some common graph algorithms and their applications in graph data analytics. We'll extend our discussion on graph data analytics to the requirement of graph deep learning and why it stands as a specialized subdomain compared to applying existing architectures.

In this chapter, we'll cover the following topics:

- Do we need graphs?
- Formalizing graphs
- Types and properties of graphs
- Graph data structures
- Traditional graph-based solutions
- The need for **representation learning**
- **Graph neural networks** (**GNNs**) and the need for a separate vertical

Do we need graphs?

The recent **artificial intelligence** (**AI**) revolution is the tip of the iceberg of a megatrend that has been impacting the computing industry for decades now. Over time, computing performance has increased exponentially against power consumed and cost; information storage costs have also decreased exponentially. To put this into perspective, while a terabyte of data can be stored in a disk costing around 100 US dollars in 2024, it would have taken more than a million dollars in the early 1990s!

Using computers and their derivative products, such as software, web applications, games, and multimedia content, has become deeply tied to our normal lifestyle. This dependence led to the need for understanding the behavior of all the interacting entities: humans, computer hardware, software such as web applications, and even organizations as a whole. The end goal was to find ways to make interactions more efficient, which could lead to unprecedented business opportunities.

Initially, given the constraints of the time, the information that was collected was less organized and the recorded truth provided a very high-level overview of systems, and only about a handful of variables within the system (for data scientists, think of data at aggregated levels, and with a small number of dimensions). At some point, someone realized computing power and data storage were cheap enough that you could record facts more granularly: not only could individual scenarios be recorded separately and more frequently, but other variables could also be recorded every time a snapshot was taken. The data revolution had begun, and stakeholders realized that by capturing and reviewing enough data about these interacting entities, a holistic picture of their behavior in the ecosystem could be drawn. The 2010s were spent commoditizing data and its products, to the point that even Series A-funded startups have adopted a data solution: be it warehousing, Elasticsearch, or recommendation engines.

Taking a step back, let's understand what data means. A data point is essentially just a factual statement. Very little discussion exists *on how the fact is represented in the data*. The de facto representation of data is tabular, and this has generally worked well for the capabilities built around the current data ecosystem. The focus on data science research and engineering has revolved around existing database architectures, which is why the tabular form of representation is the most widespread. However, the tabular form need isn't the only form of representation. The purpose of this chapter is to build a case around the graph form of representation and why graph representations can be the best option for several practical scenarios.

Graph data is represented using **nodes** (also called objects, vertices, or nouns) and **edges** (also called relationships, links, or verbs). Certain real-life scenarios necessitate emphasizing the relationships between the objects rather than just treating each object as an independent entity. Graph data structures provide us with a natural way to express these scenarios, as opposed to something such as the tabular format. Using the simple construct of treating entities as nodes and relationships as edges between two nodes, graph representations can effectively model information from a wide range of domains: from network topology to biological systems and supply chains to molecular structure.

A case study

To make this point clearer, let's consider a common question that arises in social networks. For a user (say John), we want to ascertain whether another user (say Mary) is a second-degree connection (to John). A second-degree connection simply means Mary and John have a common connection, but Mary isn't directly connected (is friends with) to John. The social media platform commonly tracks this piece of information between a pair of users and determines whether they should be recommended by the platform to connect. We'll tackle this problem from two perspectives: first using the tabular representation, and then using the graph representation.

Tabular representation

First, we need to understand what the schema of the tables in the database would be. In a typical social media platform database, there would be several tables – one for users (capturing demographic information such as age, location, date of joining, and so on), one for posts (containing details about a post made, such as the user who made the post, the contents of the post, the date of making the post, the visibility level, and so on), and many more. The table of concern for us would be something called the **connections table**. It should capture information about which users are connected directly (that is, they have a first-degree connection). The schema should go somewhat like this:

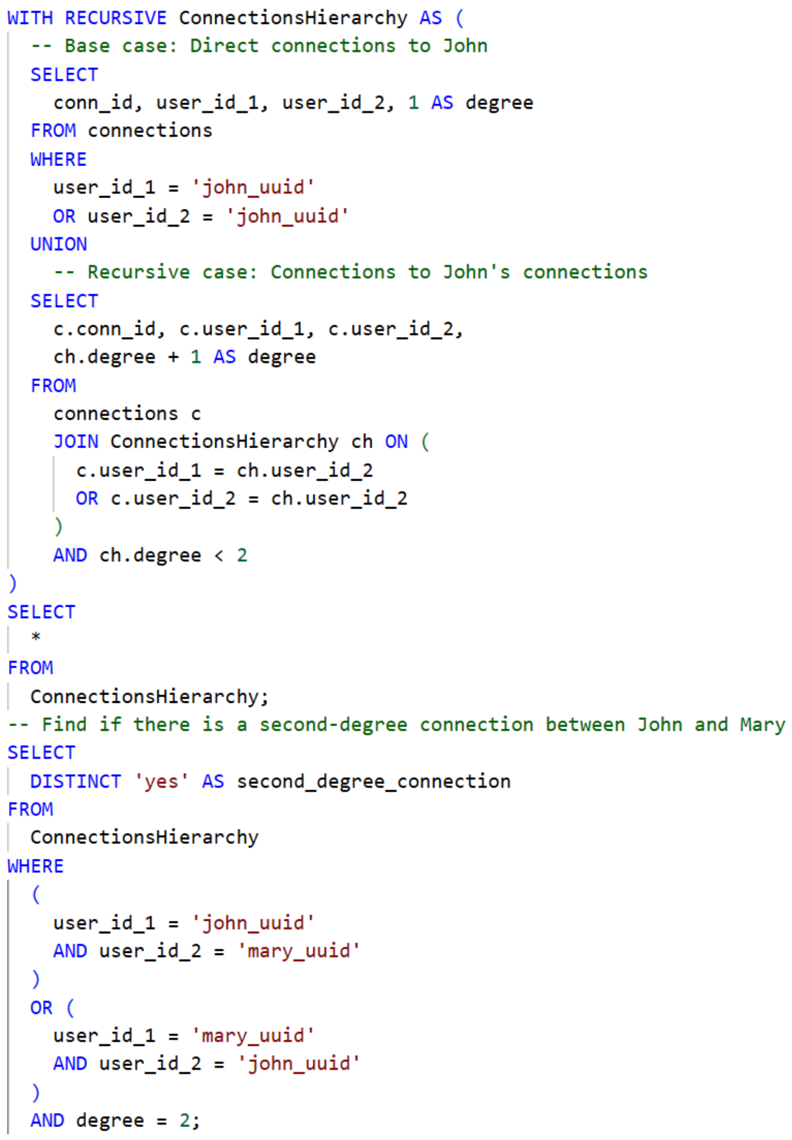
```
connections(
    conn_id: UUID,
    user_id_1: UUID FOREIGN KEY,
    user_id_2: UUID FOREIGN KEY,
    date_of_conn: TIMESTAMP,
    status_of_conn: TEXT
)
```

Table 1.1 shows a table that contains a few data points:

conn_id	**user_id_1**	**user_id_2**	**date_of_conn**	**status_of_conn**
conn_uuid_0	john_uuid	alex_uuid	2022-10-30	active
conn_uuid_1	alex_uuid	greg_uuid	2023-03-12	active
conn_uuid_2	greg_uuid	mary_uuid	2023-04-11	active
conn_uuid_3	mary_uuid	alex_uuid	2023-06-09	active

Table 1.1 – Example data stored in tabular format

To determine whether John and Mary have a second-degree connection, a SQL query similar to the following can be executed:



```sql
WITH RECURSIVE ConnectionsHierarchy AS (
  -- Base case: Direct connections to John
  SELECT
    conn_id, user_id_1, user_id_2, 1 AS degree
  FROM connections
  WHERE
    user_id_1 = 'john_uuid'
    OR user_id_2 = 'john_uuid'
  UNION
    -- Recursive case: Connections to John's connections
  SELECT
    c.conn_id, c.user_id_1, c.user_id_2,
    ch.degree + 1 AS degree
  FROM
    connections c
    JOIN ConnectionsHierarchy ch ON (
      c.user_id_1 = ch.user_id_2
      OR c.user_id_2 = ch.user_id_2
    )
    AND ch.degree < 2
)
SELECT
  *
FROM
  ConnectionsHierarchy;
-- Find if there is a second-degree connection between John and Mary
SELECT
  DISTINCT 'yes' AS second_degree_connection
FROM
  ConnectionsHierarchy
WHERE
  (
    user_id_1 = 'john_uuid'
    AND user_id_2 = 'mary_uuid'
  )
  OR (
    user_id_1 = 'mary_uuid'
    AND user_id_2 = 'john_uuid'
  )
  AND degree = 2;
```

Figure 1.1 – A SQL query being performed over tables that were introduced previously to retrieve second-degree connections

The crux of this query contains a recursive self-join operation, where each recursion level contains the connections of a certain degree. The initial filter of `user_id_1 = 'john_uuid'` OR `user_id_2 = 'john_uuid'` ensures that we only concern ourselves with users who are on some level and connected to John. Finally, by filtering by `degree = 2`, we can get the list of all users who have a second-degree connection to John.

How efficient is this approach? The worst-case time complexity can be evaluated asymptotically and expressed in **Big-O notation**. Let V be the count of users present on the social media platform and E be the count of all first-degree connections (or the number of entries in the connections table). Join algorithms have evolved, and current join operations are very efficient, but if we look at the naive approach, where two tables have lengths L and R, a join operation would have an $O(L * R)$ time complexity. Applying this logic to the preceding recursion, we'll see that the time complexity of the entire query is $O(E * E)$ or $O(E^2)$.

Graph representation

Now, let's look at the graph representation for the same problem. Let each node of the graph represent a user and each edge connecting two nodes represent a first-degree connection between the users that the connected nodes represent. An illustration using *Table 1.1* would look like this:

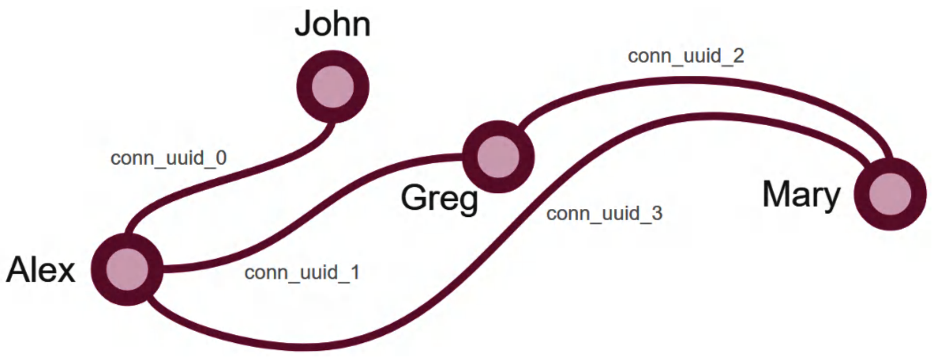


Figure 1.2 – Representing data from Table 1.1 in a graph

How do we find the answer to whether John and Mary have a second-degree connection here? We can employ an intuitive algorithm over this graph:

1. **Start from a source**: Begin at a chosen starting point, often called the *source* or **initial node** of the graph. This is your current position for exploration. For our use case, the initial node would be that of John.
2. **Explore neighbors level by level**: Visit all the neighbors of the current node before moving on to their neighbors. Imagine exploring the graph in layers, moving outward one level at a time. This ensures that you discover all nodes at a certain distance before moving farther away.
3. **Mark visited nodes**: As you visit each node, mark it as visited to avoid revisiting the same node. Use a queue to keep track of the order in which you encounter nodes. While marking the nodes, you can also keep track of how many jumps from the initial node were made to reach this node. Continue this process until you've visited all reachable nodes from the starting point.

In simpler terms, this algorithm explores the graph by gradually moving away from the starting point, checking neighboring nodes level by level, and keeping track of visited nodes to avoid duplication. It's like ripples spreading out from a pebble dropped in a pond, exploring nearby areas before moving to more distant ones. This algorithm is called **breadth-first search** (**BFS**), and it's one of the most popular graph algorithms:

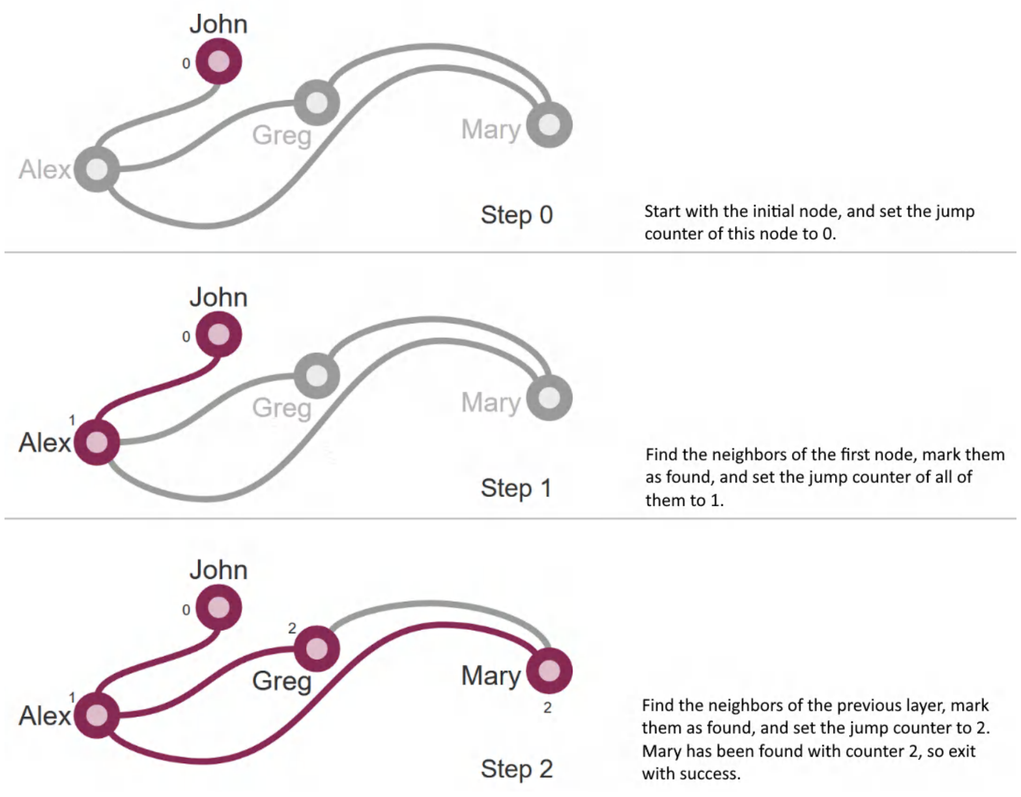

John
0
Greg
Mary
Alex
Step 0
Start with the initial node, and set the jump counter of this node to 0.
John
0
Greg
Mary
1
Alex
Step 1
Find the neighbors of the first node, mark them as found, and set the jump counter of all of them to 1.
John
0
2
Greg
Mary
1
Alex
2
Step 2
Find the neighbors of the previous layer, mark them as found, and set the jump counter to 2. Mary has been found with counter 2, so exit with success.

Figure 1.3 – Running BFS on the graph

By using this algorithm, if Mary's node is marked as visited and has a jump count of *2*, then we can safely say that John and Mary have a second-degree connection.

What's the time complexity of BFS? As mentioned previously, the number of users is assumed to be V, and the number of first-degree connections is E. Effectively, BFS touches all vertices and edges of the graph at most once, so the time complexity is simply $O(V + E)$. In a practical scenario, the number of connections far outweighs the number of users on the platform, so the time complexity can be approximated to $O(E)$.

Is $O(E)$ better than $O(E^2)$? Definitely. We can see that just by changing the perspective of approaching the problem, we achieve a much more efficient solution. To further test your understanding, think of a scenario where the problem was kept the same, except you now have to check whether John and Mary were third-degree connections instead of second-degree connections. How would the time complexities of both approaches be affected?

Graphs are useful in practice. But before we talk about how certain properties of graphs and algorithms are used to solve graph problems, we need to define a common language that we can use to refer to graphs and their properties. The following section will cover how graphs are commonly defined in mathematics, and how a simple representation can cover all the different types of data that graphs can represent.

Formalizing graphs

Graphs are a very popular concept in mathematics. In this domain, a common terminology is well accepted. Let's take a closer look.

Definition and semantics

With the argument being made for graph representations to be a relevant topic for practical problems, let's take a moment to define what a graph is. A graph is an abstract concept. Mathematically, it's generally represented as $G(V, E)$, where G is the graph, which contains a set of vertices, V, and a set of edges, E. Each element of E is a tuple, (V_1, V_2), where $V_1, V_2 \in V$, and represents a connection between the two vertices. That's all there is to the mathematical definition; how you choose to apply semantics to this is completely up to you.

In the example mentioned in the previous section, the users of the social media platform were represented by the vertices, and the connection between the two users was represented by the edges. Also, vertices and edges need not be so homogeneous. Consider the graph representation of a home network:

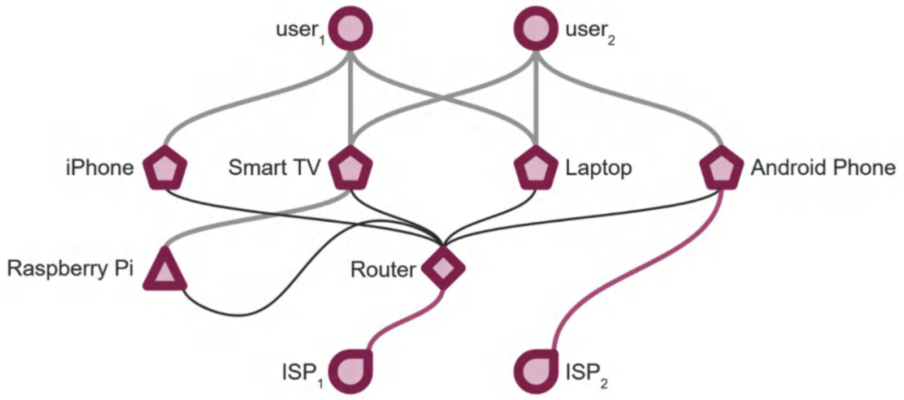


Figure 1.4 – A classic heterogeneous graph, where there are multiple types of elements represented as nodes. There is also heterogeneity in the interactions represented by edges

Here, the nodes represent all entities present in the home network, from human users to **Internet of Things** (**IoT**) devices, routers, and smart TVs. The semantics of the edges range from interaction to network communication and media streaming. Formally, the set of vertices and edges can be defined as follows:

```
V = {user_i, user_2, iPhone, Smart TV, Laptop, Android Phone, Raspberry
Pi, Router, ISP_1, ISP_2}
E = {
   (user_1,iPhone: interacts with),
   (user_1,Smart TV: interacts with),
   (user_1,Laptop: interacts with),
   (user_2,Smart TV: interacts with),
   (user_2,Laptop: interacts with),
   (user_2,Android Phone: interacts with),
   (iPhone,Router: communicates with),
   (Smart TV,Router: communicates with),
   (Laptop,Router: communicates with),
   (Android Phone,Router: communicates with),
   (Raspberry Pi,Smart TV: streams to),
   (Raspberry Pi,Router: communicates with),
   (Router,ISP_1: is connected to),
   (Android Phone,ISP_2: is connected to)
}
```

Graph representations can be further supercharged by adding more information specific to the nodes or edges. A popular way to represent this information is by using a feature vector. We'll learn more about how node and edge features can be added in future chapters.

The key takeaway here is that graphs are an incredibly powerful way of representing facts. With this definition at hand, let's try to investigate a few derived characteristics of graphs. Certain characteristics of graphs imply certain high-level facts about how the nodes and edges in the graph are organized. The next section focuses on a few such popular properties and graph types.

Types and properties of graphs

Several types of graphs have been identified, each with its unique properties, but we'll focus on the ones that are most popular. Note that these types need not be mutually exclusive, meaning a graph can be labeled as more than one type at a time.

Directed graphs

Graphs are **directed** when the edges have a one-way relationship between their connecting nodes. There are many scenarios where the relationship that's represented is unidirectional. In a graph representing a family tree, an edge might represent the relation "*is a parent of,*" and another might represent the relation "*is a pet of.*" Such relationships can't be inverted between the nodes and hold the same meaning.

Bipartite graphs

A **bipartite graph** is a type of graph whose vertices can be divided into two disjoint sets such that every edge connects a vertex from one set to a vertex in the other set. In other words, there are no edges that connect vertices within the same set. Mathematically, a graph, $G = (V, E)$, is bipartite if the vertex set, V, is partitioned into two non-empty sets, V_1 and V_2, such that every edge in E connects a vertex in V_1 to a vertex in V_2.

Bipartite graphs are often denoted as $G(V_1, V_2, E)$, where V_1 and V_2 are the two disjoint sets of vertices, and E is the set of edges connecting vertices from V_1 to V_2 One common application of bipartite graphs is in modeling relationships between two distinct types of entities, where edges represent connections or relationships between entities of different types.

Bipartite graphs occur very commonly in the wild. In e-commerce, **recommendation systems**, also known as **recommender systems** (see *Chapters 2* and *9*), are built on bipartite graph data, where the nodes consist of users and items. The users and items never interact within their own kind; only interactions between users and items exist. This interaction can be in the form of clicks or orders of items made by the user:

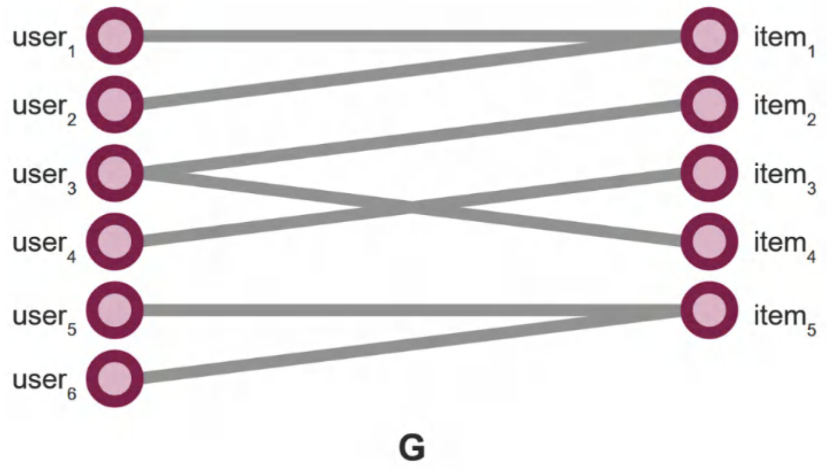


Figure 1.5 – A classic example of a bipartite graph, from an e-commerce application

Another example of a bipartite graph is the **marriage problem**, where vertices in one set represent men, vertices in the other set represent women, and edges represent marriages between couples. Another example is modeling interactions between customers and products in recommender systems.

Connected graphs

Fully **connected graphs**, also known as **complete graphs**, are graphs in which every pair of distinct vertices is connected by an edge, forming a network where each node is directly linked to every other node. They exhibit high connectivity but can become computationally intensive as the number of nodes increases.

Weighted graphs

Graphs can also be appended with additional information on both the nodes and edges. When the edges are added as scalar additional information, the graph is said to be **weighted**.

While on the topic of graph types, let's complete our discussion by introducing a few commonly noted properties of graphs. These aren't labels that are attached to graphs, as we discussed previously regarding the different types of graphs, but certain measures or attributes of graphs that are agnostic of the type of graph under concern.

Subgraphs

A **subgraph** is a graph that's formed from a subset of the vertices and edges of the original graph. More formally, let G be a graph with vertex set V and edge set E. A subgraph of G is a graph, G', such that V (G') is a subset of V and $\mathrm{E}(G')$ is a subset of E. Often, searching for subgraphs with useful properties such as being bipartite, or connected, is an important step for the bigger problem at hand.

Centrality

Centrality is a measure in graph theory that quantifies the importance or influence of a node within a network. Nodes with high centrality are more central to the network, playing a more significant role in its structure and dynamics. There are several centrality measures, each capturing different aspects of a node's importance. Here are some common centrality measures:

- **Degree centrality**: The degree centrality of a node is the number of edges connected to it (that is, the number of neighbors it has). Nodes with a high degree of centrality are well-connected and may play a crucial role in spreading information or influence.
- **Closeness centrality**: Closeness centrality measures how close a node is to all other nodes in the network. It's the reciprocal of the sum of the shortest path distances from a node to all other nodes. Nodes with high closeness centrality can quickly interact with other nodes and are often central in terms of communication efficiency.

- **Betweenness centrality**: Betweenness centrality quantifies the number of shortest paths that pass through a node. A node with high betweenness centrality has a significant influence on communication between other nodes. Nodes with high betweenness centrality act as bridges or gatekeepers in the network, controlling the flow of information.
- **Eigenvector centrality**: Eigenvector centrality considers not only the number of connections a node has but also the centrality of its neighbors. It's based on the principle that connections to high-scoring nodes contribute more to a node's centrality. Nodes with high eigenvector centrality are connected to other central nodes, making them important in the overall network structure.
- **PageRank**: PageRank is a centrality measure that's used in web search algorithms (for example, Google's PageRank). It assigns importance to a node based on both the number and quality of its incoming links. Nodes with high PageRank are considered influential as they are linked to other important nodes.

Centrality measures help identify key nodes in a network, which can be important for understanding information flow, identifying influential individuals, or targeting nodes for interventions in various applications such as social networks, transportation systems, and biological networks. Different centrality measures may be appropriate, depending on the specific context and goals of the analysis.

Community structure

In terms of graphs, **community structure** refers to dividing a network or graph into groups or clusters of nodes that are densely connected internally but have fewer connections between groups. Nodes within a community are more likely to share similar properties, interests, or functions, and identifying community structure is a fundamental aspect of analyzing the organization and dynamics of complex networks. Detecting communities in a graph is crucial for understanding the modular organization of the system and can have applications in various fields, including social network analysis, biology, and information retrieval.

Isomorphism

Isomorphism is a concept in graph theory that deals with the structural similarity between two graphs. Two graphs are considered isomorphic if a one-to-one correspondence exists between their vertices such that the adjacency relationships are preserved. In other words, the graphs are essentially the same from a structural point of view, even if the vertex and edge labels may differ.

Graph isomorphism is a fundamental problem in computer science and has applications in various areas, such as chemistry, computer-aided design, and pattern recognition. Despite its practical importance, finding a fast algorithm for graph isomorphism has proven to be a challenging problem, and it remains an open question whether such an algorithm exists with polynomial time complexity:

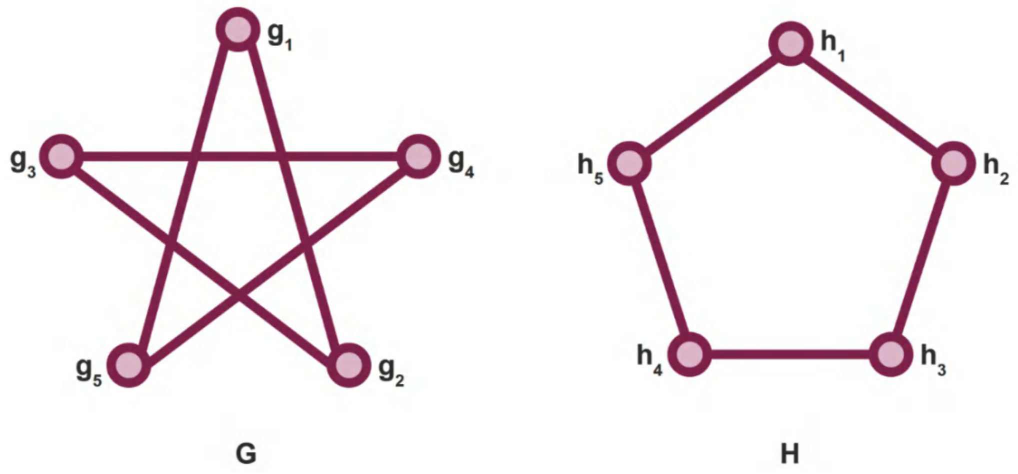


Figure 1.6 – G and H are isomorphic graphs. To understand this, notice the mapping: $g_1 \rightarrow h_1$, $g_2 \rightarrow h_2$, $g_3 \rightarrow h_3$, and so on

The properties of specialized graphs can be exploited to derive further insights into the scenarios represented by these graphs. Now, let's take a look at how graphs can be fed into machines so that they can be read algorithmically.

Graph data structures

How should we feed graph data into computer programs so that we can apply graph-based algorithms to solve problems? This will be addressed in this section. Each representation has its advantages and disadvantages, and we'll explore them from the perspective of the time complexity of determining whether an edge exists and updating the graph.

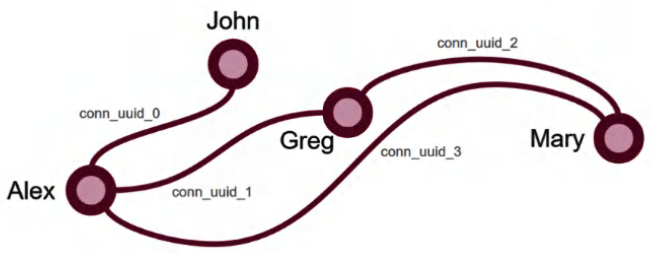
Adjacency matrix

The **adjacency matrix** aims to record the graph structure via a matrix. A matrix, say A, of size $v \times v$ is created (where v denotes the number of nodes, or mathematically, $v = |V|$). We start with all entries of A being 0. Next, if $(v_i, v_j) \in E$, then element (i, j) of A is labeled 1. If the graph is undirected, then if $(v_i, v_j) \in E$, then both elements of A, (i, j), and (j, i), are labeled 1.

The time complexity to check whether an edge exists in an adjacency matrix is $O(1)$ since it just involves checking a particular cell in the matrix. However, adding a new vertex to the graph would be difficult, and depending on the matrix implementation, it might need an entirely new initialization. Finally, the space complexity of the adjacency matrix is also large, in the order of $O(|V|^2)$. Sparse matrix implementations exist, which might reduce the space required by the matrix in memory, but they come with their own set of drawbacks. Adjacency matrices are a good choice when the graph is fairly small and static and also requires frequent lookup.

Adjacency list

Another popular data structure that's used to store graph data is an **adjacency list**. It contains a dictionary that has keys as node names, and each entry contains a list of all nodes connected to the key node with an edge. Unlike adjacency matrices, whose space complexity is $O(|V|^2)$, adjacency lists have a space complexity of $O(|V| + |E|)$. However, adjacency lists are far more flexible with efficient time complexities for many data-related tasks. Inserting vertices and edges is $O(1)$; accessing neighbors takes $O(1)$ as well, while finding whether an edge exists or not can take $O(|V|)$ time. Adjacency lists are a good choice when the graph is inherently sparse:



G

$$M = \left(\begin{array}{c|cccc} & \text{John} & \text{Alex} & \text{Greg} & \text{Mary} \\ \text{John} & 0 & 1 & 0 & 0 \\ \text{Alex} & 1 & 0 & 1 & 1 \\ \text{Greg} & 0 & 1 & 0 & 1 \\ \text{Mary} & 0 & 1 & 1 & 0 \end{array}\right)$$

L =
```
John: [Alex]
Alex: [John, Greg, Mary]
Greg: [Alex, Mary]
Mary: [Greg, Alex]
```

Figure 1.7 – For the given graph, G, M and L are the adjacency matrix and list, respectively

So far, we've talked about adjacency matrices and adjacency lists as the two most popular data structures that are used to represent graphs in algorithms. However, several other graph data structures can be used based on the problem statement. Each graph data structure has its own set of advantages and disadvantages, and this serves as a great domain on which further research can be performed.

Traditional graph-based solutions

Many computer scientists have etched their names in history by devising elegant solutions to seemingly complex problems involving graphs. However, graphs aren't just confined to the algorithm books, and graph-based problems are common in the wild. Lots of business problems and scientific research can be boiled down to graph-based problems, on which existing solutions can be implemented to generate the required output. In this section, we'll talk about the most popular problems in the domain of graphs, a few approaches to solving them, and where these problems are encountered in practical scenarios.

Searching

There are two fundamental approaches when performing a search over a graph: breadth-first and depth-first. Both are means to traverse a graph from a starting point to all nodes that can be reached from the initial node, but the differentiating factor is their approach.

In BFS, the algorithm explores a graph level by level, starting from the source vertex and visiting all its neighbors before moving on to the next level. This approach ensures that nodes closer to the source are visited before deeper nodes. On the other hand, **depth-first search** (**DFS**) explores a graph by going as deep as possible along each branch before backtracking. It explores one branch of the graph until it reaches the end before moving on to the next branch.

BFS is well-suited for finding the shortest path in unweighted graphs and is commonly used in network routing protocols and social network analysis. DFS, with its deep exploration, is useful in topological sorting, cycle detection, and solving problems such as maze exploration and puzzle-solving.

Now that we have a basic understanding of search algorithms, let's look at another class of problems, called **partitioning**. The need to understand graph partitions occurs frequently in practice.

Partitioning

In graph theory, partitioning refers to dividing the vertices or edges of a graph into disjoint subsets or components. The goal of partitioning is to group elements of the graph in such a way that certain properties are satisfied, or specific objectives are achieved. There are different types of graph partitioning, and the choice of partitioning criteria depends on the application or problem at hand:

- **Vertex partitioning**: This involves dividing the set of vertices of a graph into disjoint subsets. Vertex partitioning is done to achieve balance in terms of the number of vertices in each subset. This is a problem that's commonly encountered in load balancing in parallel computing, network design, and social network analysis.
- **Edge partitioning**: As the name suggests, this involves dividing the set of edges of a graph into disjoint subsets. Just like vertex partitioning, the objective here is to balance the number of edges or the total weight of edges in each subset. Edge partitioning problems generally find application in communication optimization in distributed computing, minimizing inter-partition communication.
- **Graph cut**: This involves partitioning a graph by removing a minimum number of edges. This is done to minimize the cut size (total weight of removed edges) while achieving certain constraints. Applications include image segmentation and community detection in social networks.

- **K-way partitioning**: This is a generalization of the aforementioned ideas and involves dividing the vertices or edges into *k* disjoint subsets. We find k-way partitioning problems where we need to achieve balance in terms of the number of elements in each of the *k* subsets, such as in domains such as **very-large-scale integration** (**VLSI**) design and parallel computing. You can read more here: `https://patterns.eecs.berkeley.edu/?page_id=571`.

Graph partitioning problems are often NP-hard, meaning finding an optimal solution may be computationally intractable for large graphs. Therefore, various heuristics, approximation algorithms, and optimization techniques are employed to find good solutions in a reasonable amount of time.

Another important problem in the domain of graphs is path optimization, and a lot of supply chain businesses and network researchers have been traditionally interested in solutions to such problems.

Path optimization

Two common problems in graph theory are finding the shortest path and finding the widest path between two nodes in a weighted graph. Their applications are obvious and involve route optimization in supply chains and social network analysis (the problem mentioned in the preceding case study can also be viewed as a shortest path problem between the two nodes, especially if a slight modification was made by needing the edges to be weighted).

The widest path problem is a variant of the shortest path problem. Formally, the problem is defined thus: given a weighted graph, the widest path problem seeks a path from a source vertex to a target vertex such that the minimum edge weight (bottleneck) along the path is maximized. This can be particularly relevant in network design or communication systems, where the goal is to maximize the capacity of the bottleneck link.

The most popular solution to find the shortest path between two nodes is **Dijkstra's algorithm**. While not diving too much into the details, here's a short step-by-step summary of the algorithm:

1. **Initialization**:
    I. Start at the source vertex.
    II. Assign a tentative distance of 0 to the source and infinity to all other vertices.
    III. Mark all vertices as unvisited.
2. **Iterative exploration**:
    I. Select the unvisited vertex with the smallest tentative distance.
    II. For the selected vertex, consider all its neighbors.
    III. Update the tentative distance of each neighbor by adding the distance from the current vertex.
    IV. If the updated distance is smaller than the current tentative distance, update it.
    V. Mark the current vertex as visited.

3. **Termination**:

    I. Repeat the iterative process until the destination vertex is visited or all vertices have been visited.

    II. The final assigned distances represent the shortest paths from the source to all other vertices.

    III. Reconstruct the shortest path by backtracking from the destination to the source using the information that was stored about the shortest distances.

In summary, Dijkstra's algorithm explores the graph in a step-by-step manner, always choosing the vertex with the smallest tentative distance. It gradually builds up the shortest paths from the source to all other vertices while marking visited vertices. The final result is a set of shortest distances and paths from the source to all other vertices in the graph. The widest path problem can also be solved by the same algorithm, but instead of maintaining the smallest tentative distance per node, we maintain the bottleneck weight (or the maximum of the minimum weights across all paths). The time complexity of this algorithm depends heavily on the implementation used, and the complexity ranges from $O((|V| + |E|)*log(|V|))$ to $O(|V|^2)$. Dijkstra's algorithm fails when the edge weights are negative, however, and is also inefficient if the graph is nearly fully connected. In such scenarios, alternatives such as the **Bellman-Ford algorithm** can be used.

Now that we have a basic understanding of what kind of analytics are performed on graph data, let's look at another powerful way of learning patterns within graph data. The core ideas of representation learning are described in the following section; we'll learn how representation learning is an important first step for solving many complex problems involving graphs.

The need for representation learning

Here, we'll introduce a new concept called **representation learning for graphs**. Let's use a small analogy to understand what this means. A typical corporate organization has several entities: employees, IT equipment, offices, and so on. All these entities maintain different types of relationships with each other: employees can be related to each other based on organizational hierarchy; one employee may use several pieces of IT equipment; several pieces of equipment, such as servers, can be networked with each other; employees and equipment can report physically or be located in a particular office, respectively; and so on.

A graph, quite rightly, seems like a natural way to represent this information, like this:

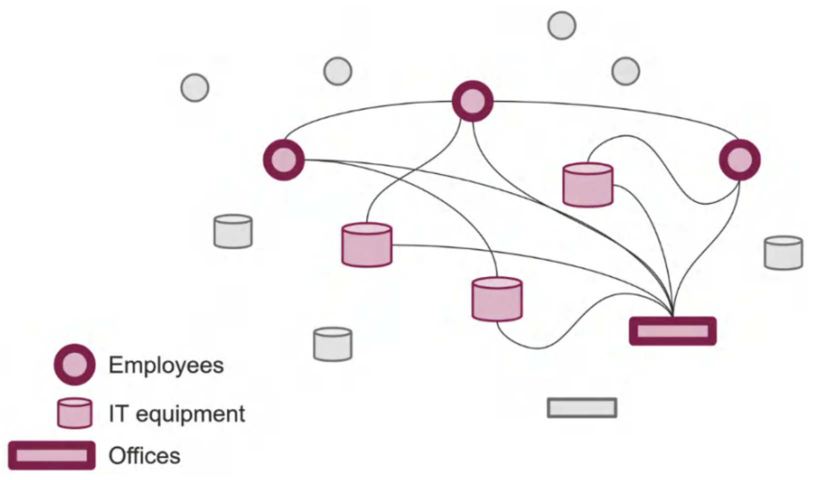
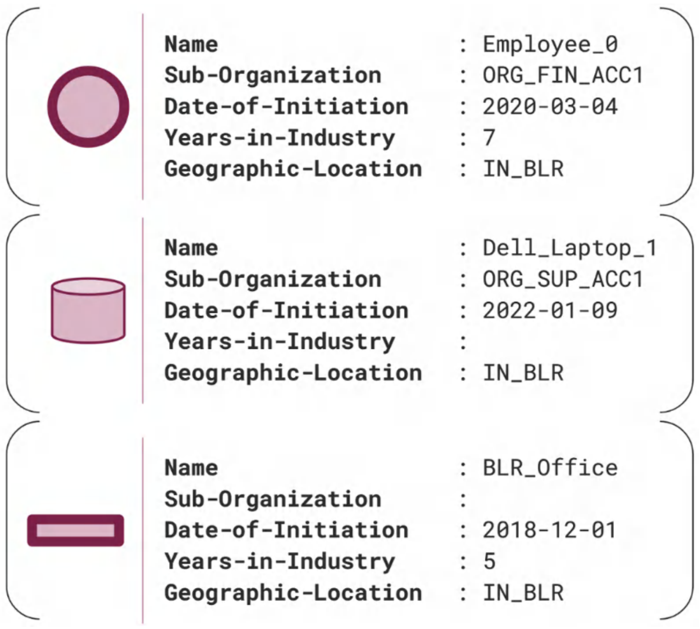


Figure 1.8 – A graph showing the different entities in an organization interacting with each other

Graphs are very visually intuitive. However, performing algorithmic calculations on graphs isn't trivial. Could we find a way to capture the characteristics as well as the relationship information of each entity as much as possible, but within some common fields? Consider a sort of unique ID card for each entity that contains the following fields: Name, Sub-Organization, Date of Initiation, Years in Industry, and Geographic Location. So, a few ID cards may look like this:



```
Name                 : Employee_0
Sub-Organization     : ORG_FIN_ACC1
Date-of-Initiation   : 2020-03-04
Years-in-Industry    : 7
Geographic-Location  : IN_BLR

Name                 : Dell_Laptop_1
Sub-Organization     : ORG_SUP_ACC1
Date-of-Initiation   : 2022-01-09
Years-in-Industry    :
Geographic-Location  : IN_BLR

Name                 : BLR_Office
Sub-Organization     :
Date-of-Initiation   : 2018-12-01
Years-in-Industry    : 5
Geographic-Location  : IN_BLR
```

Figure 1.9 – Examples of a few ID cards with the same fields for all entities

Now, if you're presented with a set of ID cards of all entities and not the preceding graph, could you answer questions such as which employee reports to whom? Alternatively, which ID card is that of an employee, which of the office, and which of a piece of IT equipment? It's not easy (or completely feasible) to determine the answers to such questions using just the information in the ID cards, but we can get close to the correct answer. To answer whether an employee is under some other employee in the hierarchy, we can check whether both employees have the same **Sub-Organization** and **Geographic-Location** values and whether the employee higher in the hierarchy has a higher **Years-in-Industry** value than the other. Guessing whether an ID card belongs to an office might simply mean checking whether the **Sub-Organization** value of the entity is blank, and so on.

While this approach sounds non-intuitive at first, using such heuristics to try to answer these practical questions based simply on the ID cards, and not the graph, can be performed efficiently by computers and modern-day implementations. The ID card here tries to *represent* the characteristics of the entities and relationships in the graph.

More formally, the ID cards in the preceding example are vector embeddings of a fixed dimension that's associated with each node of the graph. A learning algorithm is applied that tries to capture as much information present in the graph structure within these embeddings. These sets of embeddings are then used to answer difficult questions regarding the graph, instead of relying on the original graph itself.

Why should we try to learn representations? What benefits do working with embeddings present over working with the original graph in itself? Let's take a look:

- **Scalability**: Traditional graph algorithms can become computationally expensive and impractical for large-scale graphs. Representation learning allows compact embeddings to be generated that capture essential graph information, enabling more scalable and efficient computations.
- **Task flexibility**: Representation learning produces embeddings that are task-agnostic, meaning they can be used for a variety of downstream tasks. Traditional graph algorithms are often designed for specific problems, and adapting them to different tasks might be challenging. Embeddings provide a more flexible and versatile way to approach diverse tasks.
- **Incorporating node and edge features**: Many real-world graphs come with additional features associated with nodes and edges. Representation learning methods can naturally incorporate these features into the learning process, allowing for a more comprehensive understanding of the graph's characteristics. Traditional graph algorithms might not easily integrate external features.
- **Handling dynamic graphs**: Representation learning is well-suited for dynamic graphs where the structure evolves. The learned embeddings can capture temporal patterns and changes, providing a more adaptive approach compared to traditional graph algorithms, which might not inherently address temporal aspects.
- **Generalization to unseen data**: Representation learning aims to generate embeddings that generalize well to unseen data. This is particularly useful in scenarios where the graph structure is not completely known or is subject to change. Traditional graph algorithms may not generalize as effectively to new or unseen graph instances.

- **Noise robustness**: Embeddings can be more robust to noisy or incomplete graph data. Traditional graph algorithms may be sensitive to noise, but representation learning methods can learn to ignore irrelevant information and focus on the most informative aspects of the graph.
- **Integration with machine learning models**: Representation learning facilitates the integration of graph data with machine learning models. The learned embeddings can serve as input features for various machine learning tasks, allowing practitioners to leverage the power of both graph-based and non-graph-based algorithms in a unified framework.

By capturing essential information in compact embeddings, it becomes more feasible to apply machine learning techniques to graph-structured data and address a wide range of tasks efficiently.

GNNs and the need for a separate vertical

We won't dive into the details of what GNNs do or how they differ from other popular neural network architectures in this chapter. Here, we'll merely attempt to explain why there's a need to study GNNs separately from other deep learning architectures.

Before talking about the differences, we must discuss the similarities. GNNs are an architecture choice that's specialized for processing graph data and outputting representations or node embeddings. Similar to how convolutional networks are fundamental for reading pixel data, the set of architectures under GNNs are optimized for reading graph data. GNN-based learning tasks follow the same trajectory as other deep learning solutions: to iteratively optimize the parameters of the model so that a loss function can be minimized. In the case of GNNs, the loss function often tries to capture and preserve meaningful information about the graph structure.

Now, let's have a look at the differences, and why GNNs require special attention:

- **Irregular graph structures**: Graphs can have irregular and varying structures, with nodes having different numbers of neighbors. This irregularity poses challenges for traditional deep learning architectures, which often assume fixed input sizes. For example, in a social network, individuals may have varying numbers of connections (friends, followers, and so on).
- **Permutation invariance**: The order of nodes in a graph shouldn't affect the output of a GNN. Achieving permutation invariance is crucial to ensure that the model can generalize across different node orders. A good example of permutation invariance is in molecular graphs in chemistry. The properties of a molecule depend only on its structure, not the order in which the atoms are labeled.
- **Graph isomorphism problem**: Determining whether two graphs are isomorphic is a computationally complex problem. GNNs need to be able to capture and differentiate between different graph structures.
- **Node and graph classification**: GNNs often need to perform tasks such as node classification or graph classification. These tasks require the model to capture both local and global information from the graph, which can be challenging.

- **Handling different types of edges**: Graphs may have different types of edges, each representing a different kind of relationship between nodes. GNNs need to be able to model and leverage this heterogeneity in edge types.
- **Pooling and aggregation**: Aggregating information from neighbors in a graph is a fundamental operation in GNNs. Designing effective pooling and aggregation strategies to capture important information while avoiding information loss or redundancy is a challenge. A good example is when recommendation systems pool information from the user's friends' preferences. The representations should capture this pooling idea as well.
- **Capturing long-range dependencies**: GNNs need to capture long-range dependencies in graphs. Unlike sequential data in **recurrent neural networks** (**RNNs**), where dependencies are often local, graphs may have dependencies that span long distances.
- **Interpretable representations**: Understanding and interpreting the learned representations in GNNs is crucial, especially in applications where interpretability is essential. Interpretability is crucial in medical diagnosis with electronic health records. GNNs can be used to analyze patient records where nodes represent different health parameters. Interpretable representations are crucial for healthcare professionals to understand and trust the model's decision-making process in diagnosing diseases or predicting patient outcomes.

GNNs provide a lot of the benefits that machine learning approaches have in terms of boosting our capabilities with tabular data, but in terms of graph data. There are several important implications of using GNNs as an intermediary step to solve relevant problems related to graphs. We'll explore more such topics in the coming chapters.

Summary

In this chapter, we covered the foundational concepts in graph learning and representation. We began with motivating examples of how graph structures naturally capture relationships between entities, making them a powerful data representation. Then, formal definitions of graphs, common graph types, and key properties were discussed. We also looked at popular graph algorithms such as searching, partitioning, and path optimization, along with their real-world use cases.

A key idea presented here was the need for representation learning on graphs. Converting graph data into vector embeddings allows us to leverage the capabilities of machine learning models. Benefits such as scalability, flexibility, and robustness make graph embeddings an enabling technique.

Finally, we justified the need for specialized GNN architectures. Factors such as irregular structure, permutation invariance, and complex operations such as aggregation and pooling necessitate tailored solutions. GNNs open up new possibilities for learning from relational data across domains.

In the next chapter, we'll discuss how graph learning is applied in practice. There are several levels to graph learning and representation, and all of them have business and academic significance. The topics covered here will act as a foundation for your understanding of the concepts in the chapters that follow.

2
Graph Learning in the Real World

In the ever-evolving landscape of data science and **machine learning** (**ML**), the transformative power of graph-based learning has emerged as a pivotal force in unraveling the complexities of real-world phenomena. From social networks and transportation systems to biological interactions and e-commerce, a multitude of intricate systems can be abstracted and analyzed through the lens of graphs. This chapter delves into the fascinating realm of graph learning in the real world, where we explore the compelling idea that many intricate real-world problems can be effectively translated into node-, edge-, and graph-level prediction tasks.

Graphs, composed of **nodes** representing entities and **edges** capturing relationships between them, provide an intuitive framework for modeling interconnected structures. By harnessing the inherent relationships encoded in graphs, we gain a powerful tool for understanding the dynamics and patterns underlying diverse domains. The focus will be on harnessing the predictive potential embedded in the topology and structure of graphs, thereby enabling solutions to problems ranging from **recommendation systems** and fraud detection to drug discovery and urban planning.

At the heart of graph learning are *nodes*, the elemental entities in a network. Nodes can be anything from social network users to biological molecules and cities in a transportation system. By learning and predicting node attributes or behaviors, we can build applications such as personalized recommendations and targeted marketing.

Edges, representing the relations between nodes, add another dimension of complexity. Edges can capture social ties, biological interactions, or network connections. By learning and predicting edge properties or outcomes, we can gain insights into how entities relate to each other. This is important for tasks such as link prediction, anomaly detection, and network optimization.

Beyond the individual nodes and edges, graph-level predictions offer a global perspective. By learning and analyzing the overall structure and features of a graph, we can discover patterns, communities, and phenomena that emerge from the network. This is useful for scenarios such as urban planning, where the design of city infrastructure can affect various aspects of daily life.

In this chapter, we'll explore the following topics:

- Node-level learning
- Edge-level learning
- Graph-level learning
- Real-world applications

Node-level learning

Node-level learning is the task of learning and predicting attributes or behaviors of individual nodes in a graph. Depending on the type and range of the target variable, node-level learning can be categorized into four subtasks: **node classification**, **node regression**, **node clustering**, and **node anomaly detection**.

Node classification

Node classification in graphs is an ML task that aims to assign labels or categories to nodes based on their features and connections. For example, in an e-commerce graph, we can classify users into different preference groups based on their interactions with items. To do this, we need to extract features from both nodes and edges, such as demographic information, item attributes, popularity, reviews, frequency of purchases, and time spent on items. These features capture the characteristics and preferences of users and items, as well as the strength and nature of their interactions. By training an ML model on a subset of labeled nodes, we can then predict the labels for the unlabeled nodes. This way, we can segment users into groups such as *Health Conscious*, *Tech Enthusiasts*, or *Fashion Lovers* and provide them with more relevant and personalized recommendations.

Take this graph:

$$G(V,E)$$

Here, V is the set of nodes, E is the set of edges, and each node V_i possesses a feature vector X_i.

The task of node classification aims to predict the label for each node V_i based on its features and the graph structure. Mathematically, this can be represented as follows:

$$f(Xi, G) \rightarrow y_i$$

Here, *f* is the node classification function that considers the features of the node *Xi* and the structure of the graph *G* to produce the predicted label y_i. The function *f* is typically learned from labeled examples in a training dataset, where *nodes* are associated with ground-truth labels. The goal is to generalize this learning to accurately classify nodes with unknown labels in unseen data.

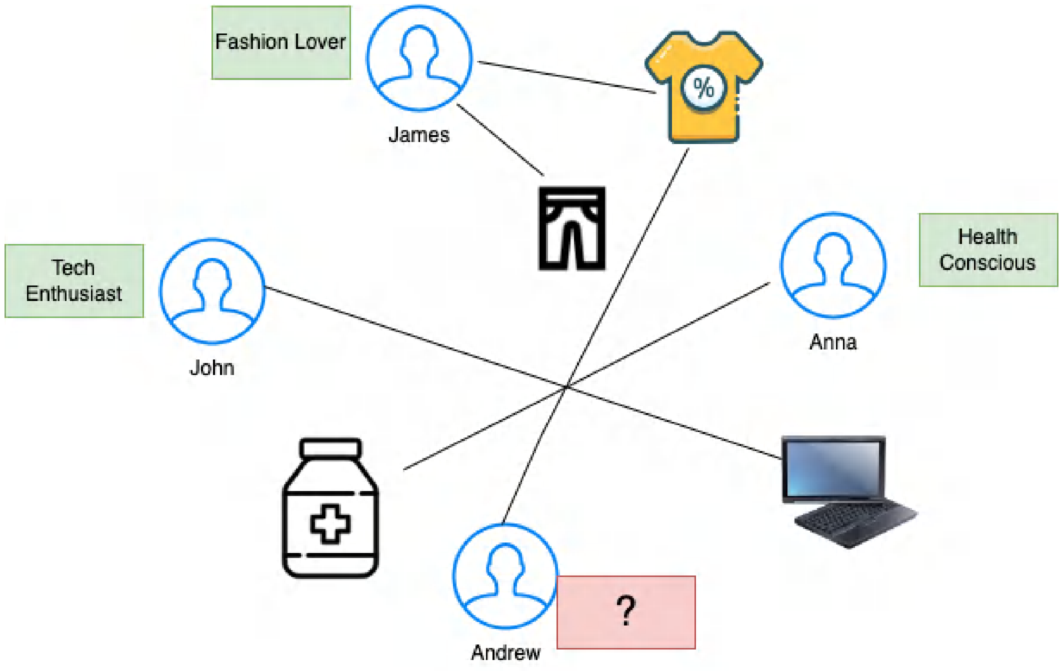


Figure 2.1 – Node classification task for e-commerce

In *Figure 2.1*, a labeled e-commerce graph, our objective is to determine the category to which Andrew belongs.

Some other real-world examples of node classification tasks include the following:

- **Information retrieval**: Node classification can be used to categorize web pages, documents, queries, or users based on their content, structure, and interactions. For example, node classification can help classify queries into different intents on an e-commerce query graph or classify web pages into different topics on a web graph.
- **Social network analysis**: Node classification can be used to identify the roles, communities, or interests of users or entities in a social network based on their attributes, behaviors, and relationships. For example, it can help us detect fake or malicious accounts on Twitter.

- **Bioinformatics**: Node classification can be used to infer the functions, interactions, or properties of biological molecules or cells based on their features and associations. For example, node classification can help predict protein functions on a protein-protein interaction network or identify cell types on a single-cell gene expression network.

Next, let's look at what happens when we have a continuous value to predict.

Node regression

In contrast to classification, **node regression** involves predicting numerical values associated with individual nodes. This task is particularly useful in scenarios where understanding the quantitative aspects of nodes is crucial. In an e-commerce system, one possible use case is to predict the **lifetime value** (**LTV**) of customers based on their interactions with products and other features. For example, you can model the e-commerce platform as a bipartite graph, where customers and products are *nodes*, and purchases, ratings, or views are *edges*. You can extract features from both nodes and edges, such as customer demographics, product attributes, historical purchase patterns, popularity, and strength of interaction. Then, you can train a regression model to predict the LTV of each customer, which can help you segment your customers and optimize your marketing strategies.

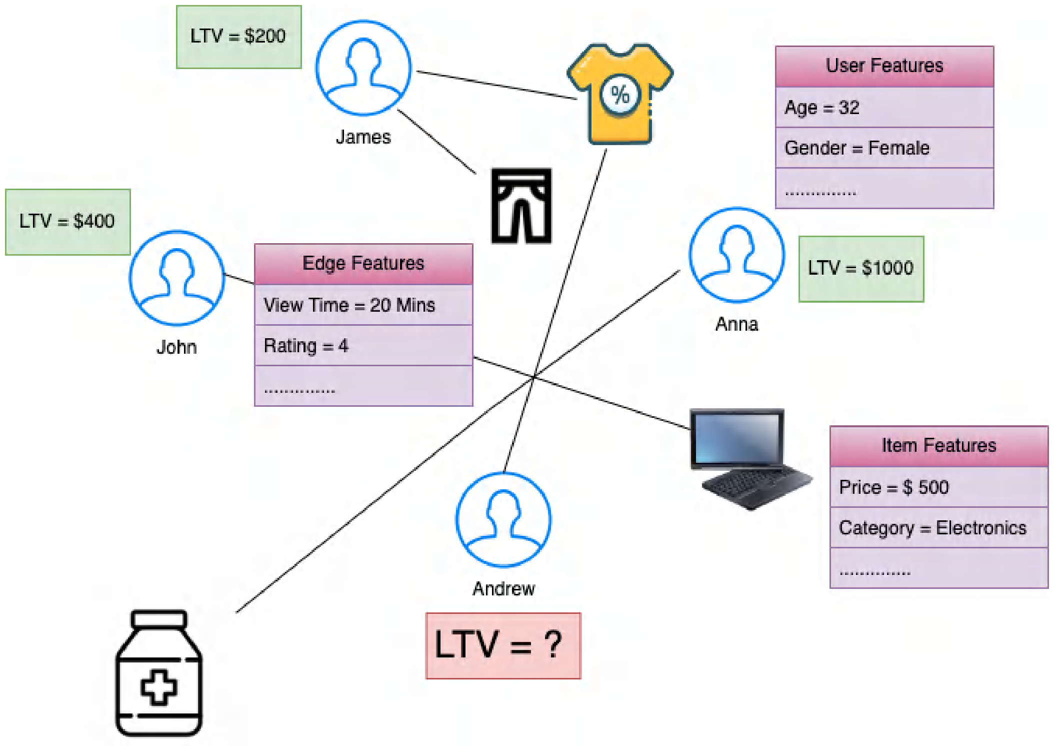


Figure 2.2 – Node regression for e-commerce LTV prediction

Figure 2.2 illustrates a labeled e-commerce graph for predicting LTV using node regression. The graph contains nodes representing users (James, John, Anna, Andrew), items (clothing and electronics), and various features associated with users, items, and their interactions. The goal is to predict Andrew's LTV based on the graph structure and available features.

Some other real-world examples of node regression include the following:

- **Product demand**: Given a product graph with products as *nodes* and similarity between products as *edges*, node regression can forecast demand for a product based on the sales of related products. This can assist inventory and supply chain management.
- **Estimating property values in real estate networks**: Nodes in a real estate graph could represent individual properties. Node regression can predict property values based on features such as location, size, amenities, and recent property sales in the neighborhood.

Let's now see where we can leverage clustering techniques in graph systems.

Node clustering

Node clustering aims to group nodes with similar characteristics or connectivity patterns. By identifying communities within the graph, this task provides a deeper understanding of the inherent structures and relationships. In a citation network, for instance, researchers working on similar topics may form distinct clusters, unveiling communities of interest.

Continuing with the e-commerce example, node clustering can be employed to group customers based on their purchase behavior and preferences. This approach facilitates a deeper understanding of customer segments, enables targeted marketing campaigns, and enhances the provision of personalized recommendations. By utilizing a graph clustering algorithm, we can identify clusters of customers who exhibit similar purchasing patterns or engage with items that are commonly bought by similar customer profiles. Node clustering can serve as a valuable tool for tailoring marketing strategies to specific customer segments and refining recommendation systems for a more personalized shopping experience.

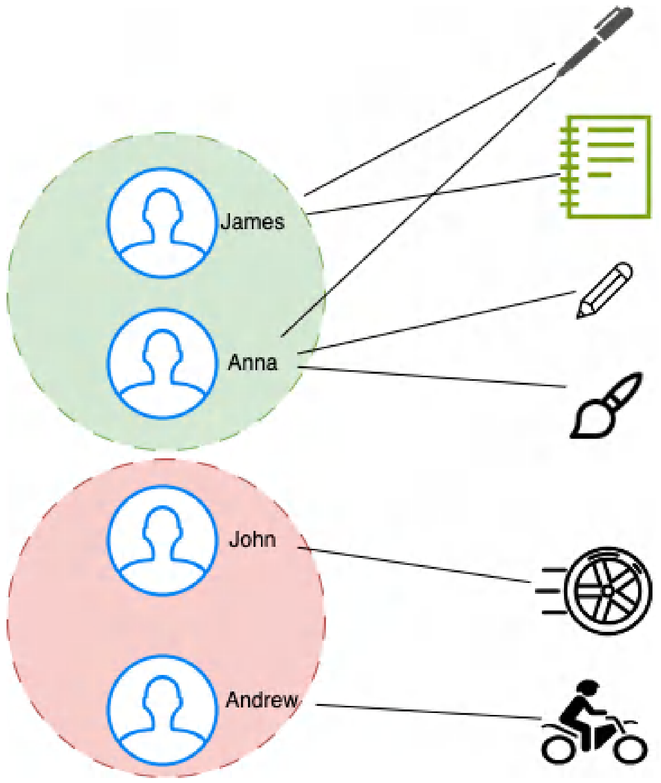


Figure 2.3 – Node clustering for e-commerce customers based on purchase behavior

In *Figure 2.3*, we see how clustering can be applied at the node level to segregate users based on their purchase history.

Some other real-world examples of node clustering involve the following:

- **Citation networks**: In academic citation networks, *nodes* can represent academic papers, and *edges* can indicate citations. Clustering papers based on similar topics, keywords, or citation patterns can assist researchers in literature review and identifying research trends.
- **Internet of things (IoT) networks**: In an IoT network, *nodes* can represent connected devices, and *edges* can indicate communication links. Clustering devices based on their functionality, usage patterns, or compatibility can aid in optimizing network traffic, resource allocation, and identifying potential security threats.

At times, we might be interested in identifying anomalies in our graph data. Let's see how!

Node anomaly detection

Node anomaly detection in graph learning is the task of identifying graph nodes that deviate significantly from the normal patterns. For instance, consider an online retail platform where users create accounts, browse products, and make purchases. Node anomaly detection could be applied to pinpoint users who display unusual purchasing behavior, such as excessively high-frequency transactions, unusually large shopping cart sizes, or sudden changes in purchasing habits. Detecting these anomalies is essential for flagging potentially fraudulent activities and ensuring a secure and trustworthy online shopping experience for genuine users.

Some other real-world examples of node clustering tasks include the following:

- **Network intrusion detection**: In a computer network, identifying nodes (computers or devices) that exhibit unusual communication patterns, such as a sudden increase in data transfers or suspicious access attempts, helps in detecting potential security threats or intrusions.
- **Telecommunication networks**: Nodes in a telecommunication network can be mobile devices or communication towers. Anomaly detection can help identify abnormal call patterns, unexpected roaming behavior, or sudden spikes in network traffic.
- **Financial fraud detection**: In financial transaction networks, anomaly detection can identify unusual patterns that may indicate fraudulent activities, such as money laundering or insider trading.
- **Cybersecurity**: Identifying compromised user accounts or devices in a network by detecting unusual access patterns or data transfer behaviors.

Next, let’s dive into edge learning – the nexus that links nodes and defines relationships.

Edge-level learning

Edge-level learning is a branch of graph ML that focuses on predicting the properties or labels of the edges in a graph, based on the features of the nodes and edges, and the structure of the graph. Edge-level graph learning can be useful for tasks such as link prediction, recommendation systems, fraud detection, and social network analysis.

Link prediction refers to the problem of predicting missing or future edges/links in a graph or network. Given a snapshot of a network, the goal is to estimate the likelihood of an edge forming between two nodes based on the existing graph structure and node attributes.

In an e-commerce graph, link prediction can be used to predict new edges between users and products that represent potential future purchases. Specifically, we can predict which products a user may be interested in purchasing, based on their previous interactions as well as similar users’ purchase patterns.

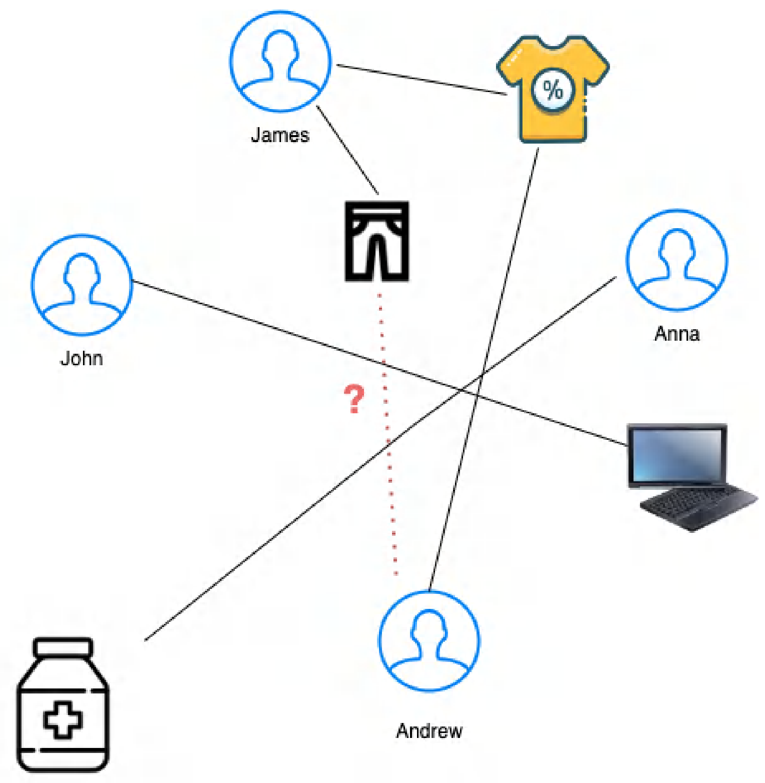


Figure 2.4 – Link prediction for e-commerce user-item recommendation

In *Figure 2.4*, we see that given graph data for **user-item interaction**, we can use link prediction task to identify if suggesting a particular product to a particular customer will lead to conversion.

Some real-world applications for link prediction include the following:

- **Recommendation systems**: Link prediction can be used to recommend products, services, or content to users based on their preferences, behavior, or feedback. For example, in a movie streaming service, where *nodes* represent users and movies, and *edges* represent explicit/implicit ratings, link prediction can be used to estimate the rating that a customer would give to a movie they haven't streamed yet.
- **Social network analysis**: Link prediction can be used to analyze the structure and dynamics of social networks, such as finding communities, influencers, or potential friends. For example, in a social network graph, where *nodes* represent users and *edges* represent friendships or interactions, link prediction can be used to predict the strength or intimacy of the relationship between two users, or the probability that two users would connect.

Now, let's explore the distinctions between link prediction and edge classification tasks.

Edge classification

Edge classification is the task of predicting a discrete label for each edge in a graph, such as the type, category, or status of the edge. For example, in an e-commerce graph, where *nodes* represent products and customers and *edges* represent transactions or ratings, edge classification can be used to classify the type of relationship represented by each edge as either casual browsing, serious interest, or purchase intent. This will allow us to understand user behavior and intent better.

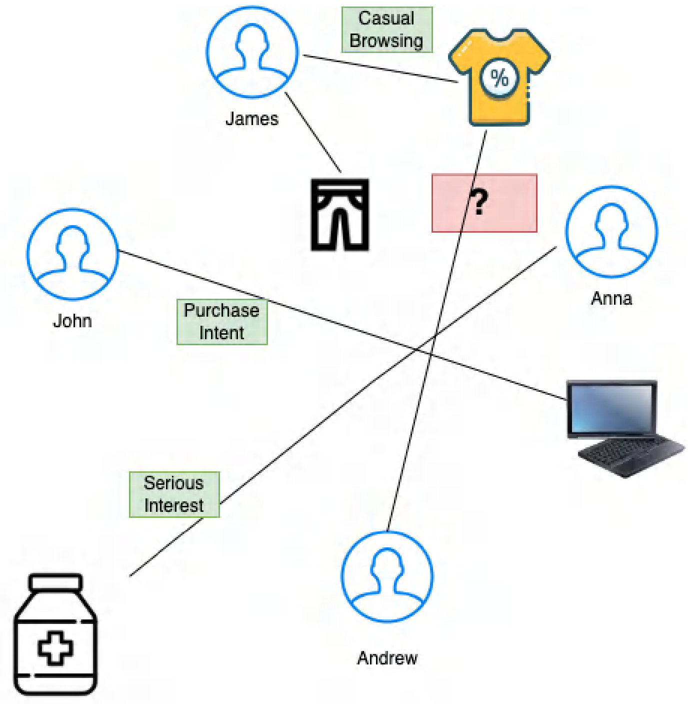


Figure 2.5 – Edge classification for e-commerce session intent

In *Figure 2.5*, the utilization of edge classification is illustrated, demonstrating how users can be categorized into different segments according to their purchase history.

Some real-world examples of edge classification are as follows:

- **Publication citation**: In a publication citation network, where *nodes* represent papers and *edges* represent citations, edge classification can be used to classify the edges as positive or negative, indicating the sentiment or tone of the citation
- **Social network**: In a social network, edge classification can be used to classify the edges as strong or weak, indicating the strength or intimacy of the relationship between the users

Similar to nodes, regression can also be performed on graph edges.

Edge regression

Edge regression is the task of predicting a continuous value for edges in a graph, such as the weight, strength, or similarity of the edge. In the e-commerce case, edge regression can be used to estimate the amount of time a customer will spend before making a purchase decision for a particular product based on historical interactions of users with various similar items, along with other user and item features.

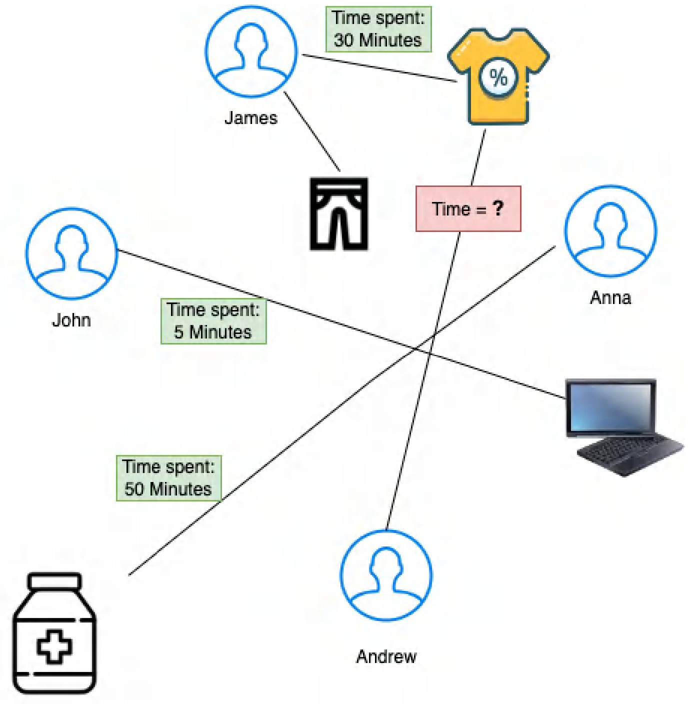


Figure 2.6 – Edge regression for predicting time between discovery and purchase

Figure 2.6 demonstrates how we can use edge regression to predict continuous labels, such as time spent on a particular product.

Some real-world examples of edge regression include the following:

- **Predicting traffic flow between locations**: In a road network graph, edge regression can predict the traffic volume between two locations based on historic flows and connectivity patterns. This can assist in traffic optimization.
- **Predicting product co-purchase rates**: In an e-commerce graph, edge regression can forecast how frequently two products might be bought together in the future based on past co-purchase data.

Edge learning extends beyond supervised techniques; it can also be applied to unsupervised learning approaches.

Edge clustering

This is the task of grouping the edges in a graph into clusters based on their features or labels, such that the edges within a cluster are more similar to each other than to the edges in other clusters. For example, in the e-commerce graph, **edge clustering** can be used to identify groups of edges that share common patterns, such as frequent or high-value purchases.

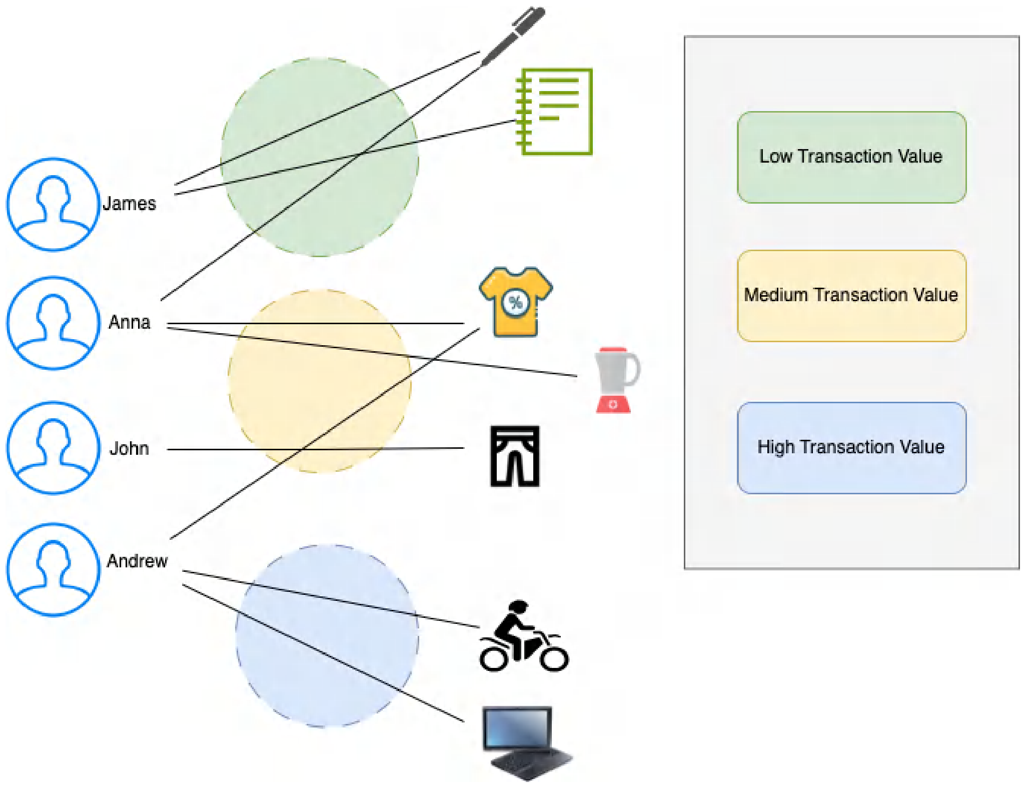


Figure 2.7 – Clustering purchases based on transaction value

Figure 2.7 shows how we can cluster edges based on transaction values to put near-similar transactions in the same bucket.

Some other applications of edge clustering are as follows:

- **Clustering financial transactions between accounts**: Edge clustering can group similar types of monetary transfers or flows between accounts, helping identify suspicious patterns
- **Clustering hyperlinks between web pages**: Edge clustering can categorize hyperlinks between web pages into groups such as navigational links, commercial links, affiliated links, and so on

Similar to nodes, we can have outlier edges that we might need to detect.

Edge outlier detection

Edge outlier detection is the task of identifying the edges in a graph that deviate significantly from the normal patterns in the graph, such as the edges that have unusual features, labels, or connections. In an e-commerce graph, edge outlier detection can be used to detect anomalies, such as fraudulent transactions and fake reviews. The following list contains some applications for edge-level outlier detection.

Some applications of edge outlier detection include the following:

- **Detecting fraudulent fund transfers**: In a banking transaction graph, identifying anomalous high-value transfers between unrelated accounts as outliers can reveal potential fraud
- **Detecting unauthorized network access**: In an enterprise network graph, edge outliers may represent rare connections between devices that indicate malicious actors or compromised devices
- **Detecting stock market manipulation**: In a stock trading network, outlier edges between accounts may reveal patterns of illegal trade collusion or pump-and-dump schemes

As we delve into the intricacies of edge learning in the preceding section, it becomes evident that its applications extend beyond conventional boundaries. Next, let's progress to the uppermost layer in the learning space, focusing on learning at the graph level.

Graph-level learning

Graph-level learning refers to ML tasks and techniques that operate at the level of entire graphs rather than individual nodes or edges within a graph. Graph-level learning focuses on generating predictions, classifications, or insights based on the entire graph or a subgraph structure.

Graph-level prediction

The goal here is to make predictions or classifications for the entire graph rather than individual nodes or edges. For example, given a time-specific user-item interaction graph for e-commerce, we can predict any special events, or patterns in general, based on graph-level learning. In urban planning and transportation management, **graph-level prediction** can be employed to forecast traffic flow across an entire road network. This can enable the optimization of traffic signal timings, route planning, and infrastructure development.

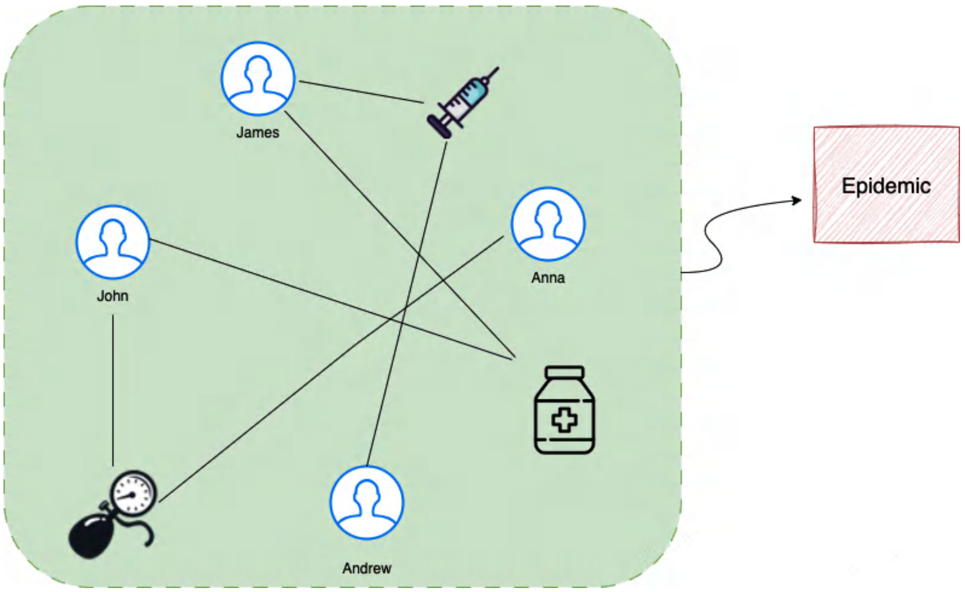


Figure 2.8 – Event prediction based on a snapshot of an e-commerce graph

Figure 2.8 shows how we can use the entire graph to make certain predictions at the graph Level.

Graph-level representations

Learning **graph-level representations** involves capturing essential features and characteristics of the entire graph. Techniques such as **graph neural networks** (**GNNs**) are employed to aggregate information from individual nodes and edges to create a numerical representation of the entire graph.

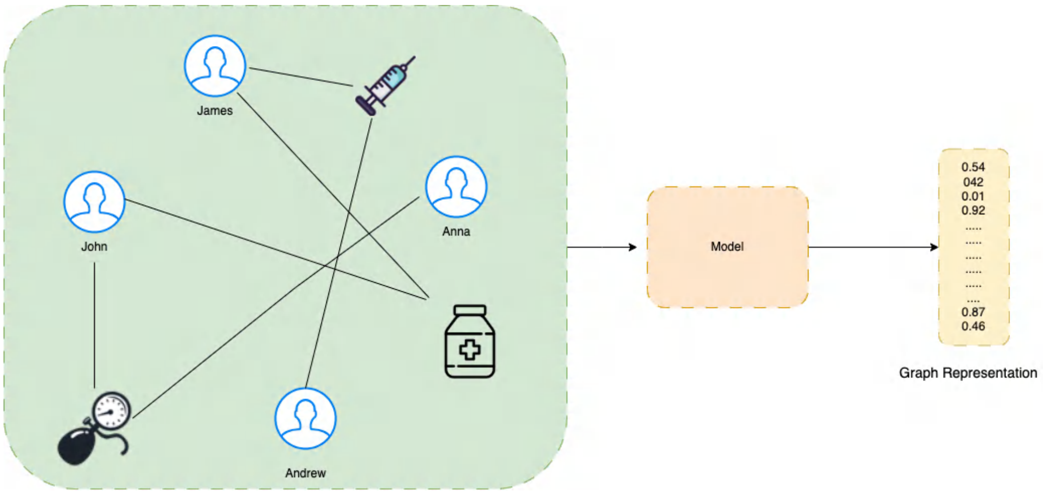


Figure 2.9 – Representing a graph with a vector of floats

Figure 2.9 shows how we can use modern deep learning techniques to represent the entire graph as an embedding vector. These representations can further be used to build models for downstream tasks such as **regression**, **classification,** or **clustering**.

Let's take one sample use case of social network analysis for product recommendation through the entire pipeline:

1. **Data collection and graph construction**: Collect user data (*nodes*) and their interactions (*edges*) from a social network platform. Construct a graph where users are nodes and friendships or interactions are edges. Include user attributes (age, interests) as node features and interaction types as edge features.
2. **GNN model**: Choose a GNN architecture (e.g., a **graph convolutional network** (**GCN**) or **GraphSAGE**). Define the number of GNN layers and their dimensions.
3. **Node-level representation learning**: For each node, aggregate information from its neighbors using the GNN. This process creates a learned embedding for each user that captures both their attributes and network structure.
4. **Graph-level representation**: Use a readout function (e.g., sum, mean, or max pooling) to aggregate node embeddings. This step produces a single vector representing the entire graph.
5. **Training process**: Define a task-specific loss function (e.g., binary cross-entropy for product recommendation). Use backpropagation to update GNN parameters, optimizing for the graph-level task.
6. **Graph embedding output**: The trained model now outputs a fixed-size vector (e.g., 128 dimensions) for any input graph. This vector captures the global properties of the social network.
7. **Downstream task – product recommendation**: Use the graph embedding as input to a classifier that predicts whether a product will be popular in the network. Train this classifier on historical data of product successes and failures.
8. **Deployment and inference**: For a new product, generate the current social network graph embedding. Feed this embedding into the trained classifier to predict the product's potential popularity.

As we have seen, graph-level learning is crucial in various applications such as bioinformatics, social network analysis, chemistry, and recommendation systems. Techniques used in graph-level learning often leverage the hierarchical and relational nature of graphs to capture complex dependencies and make predictions at a higher level of abstraction.

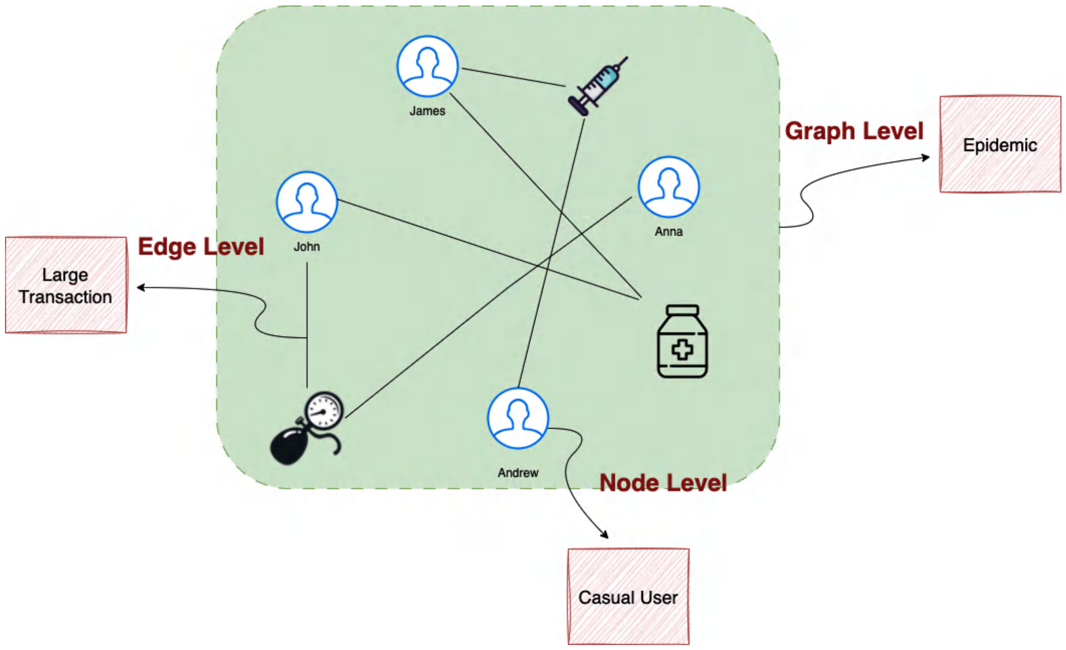


Figure 2.10 – Node-, edge-, and graph-level learning

Figure 2.10 consolidates all three learning levels—*node*, *edge*, and *graph*—onto the same canvas. This illustrates how graph learning operates across the different levels. Each level represents a distinct aspect of the data within the graph structure, with specific applications at each level.

Real-world applications

In this section, we will explore some areas where graph learning is actively being applied.

Recommender systems

Graph learning has emerged as a powerful tool in the field of recommendation systems, enhancing their capabilities and effectiveness. Recommendation systems aim to predict user preferences and provide personalized suggestions, and graph learning leverages the inherent relational structure of data to achieve this more efficiently.

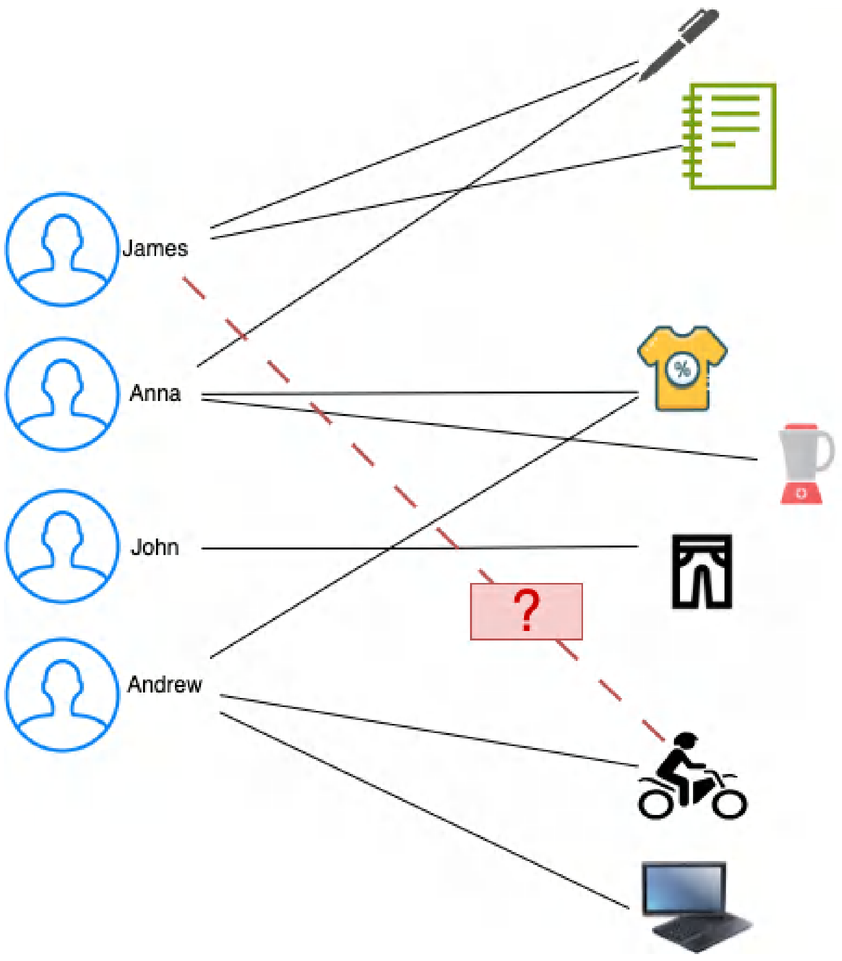


Figure 2.11 – User-item link prediction for recommendation

Figure 2.11 shows how we can translate a user-item affinity task for an e-commerce recommendation to a link prediction problem. This task is an example of one area, among others, in recommendation systems where graph learning can play a significant role.

User-item graph representation

Graph learning enables the representation of users and items as nodes in a graph, with edges indicating interactions or relationships between them. This representation captures the complex dependencies and connections in user-item interactions, allowing for a more nuanced understanding of user preferences.

For example, in a social network, users and posts can be represented as *nodes* in a graph, where *edges* indicate interactions, such as likes, comments, or shares. Graph learning can then capture the relationships between users and posts for personalized content recommendations.

Implicit and explicit feedback

Recommendation systems often deal with both implicit and explicit feedback. Graph learning can effectively model implicit feedback, such as clicks, views, and dwell time, by incorporating the graph structure to capture the relationships between users and items. This helps in making accurate predictions even when explicit feedback is sparse.

Neighborhood-based recommendations

Graph-based recommendation systems leverage the concept of **neighborhood-based learning**. By analyzing the local structure around a user or an item in the graph, these systems can recommend items that are similar to those already interacted with or liked by the user or other similar users.

Consider an example of a movie recommendation system. A user's preferences can be inferred by analyzing the movies liked or watched by users in their neighborhood (users with similar tastes). Graph learning identifies these local structures to suggest movies.

Heterogeneous graphs

In real-world scenarios, recommendation systems often deal with heterogeneous information, including users, items, and various interactions. Graph learning can handle **heterogeneous graphs**, where nodes represent different types of entities, allowing for more comprehensive modeling of relationships and preferences.

For example, in a scholarly paper recommendation system, nodes could represent authors, papers, conferences, and keywords. Heterogeneous graph learning captures relationships between these entities, allowing for personalized recommendations based on the user's research interests.

Cold start problem

The **cold start problem** occurs when a new user or item has a limited interaction history, making it challenging to provide accurate recommendations. Graph learning can alleviate this issue by leveraging the graph structure to identify and recommend items based on similar users or items, gathering learnings from multi-level hops, even in the absence of direct interactions.

Temporal dynamics

Graph learning can be extended to capture **temporal dynamics** in recommendation systems. By incorporating time-stamped edges in the graph, the model can adapt to changing user preferences over time, improving the accuracy of recommendations for evolving user behaviors.

For example, in a news recommendation system, a graph with articles and users can incorporate time-stamped edges. Graph learning can consider the evolution of user interests over time, ensuring that recent interactions have more influence on recommendations than older ones.

We will be exploring graph learning methods to solve various recommendation system problems in *Chapter 9*.

Knowledge graphs

Knowledge graphs are powerful representations of structured information that capture relationships and entities in a structured format. Graph learning can significantly enhance the capabilities of knowledge graphs, providing more nuanced insights, efficient querying, and improved reasoning.

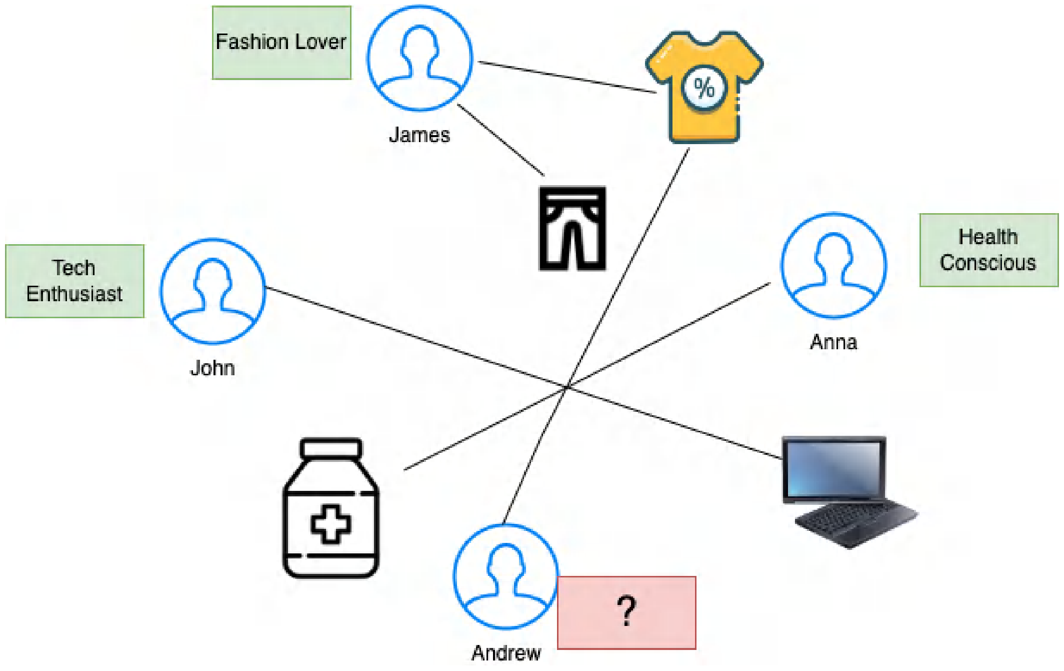


Figure 2.12 – Knowledge graph for e-commerce

Next, we will look at various aspects of knowledge graphs where graph learning can play a crucial role.

Entity and relationship embeddings

Graph learning techniques, such as GNNs, can generate embeddings for entities and relationships in a knowledge graph. These embeddings capture the latent features of entities and relationships, enabling more effective representation and understanding of the underlying semantics. For example, in a medical knowledge graph, graph learning models can generate embeddings for diseases, symptoms, and treatments. The model captures latent features, enabling a more nuanced understanding of relationships, such as the association between specific symptoms and diseases.

Link prediction

Graph learning is instrumental in link prediction tasks within knowledge graphs. By analyzing the existing structure, graph-based models can predict missing relationships between entities, helping to complete the graph and discover implicit connections.

Consider an example of a scientific knowledge graph representing academic collaborations. Graph learning predicts missing links by identifying potential collaborations between researchers who have not previously collaborated but share common research interests.

Semantic similarity and entity resolution

Graph learning can help with determining semantic similarity between entities. By considering the graph structure, the model can identify related entities, facilitating tasks such as entity resolution, where different records referring to the same entity are linked together.

For example, in a customer data knowledge graph for a retail company, graph learning can identify semantic similarity between customer profiles, aiding entity resolution by linking different records that refer to the same customer across multiple databases.

Knowledge graph completion

Graph learning can help address incompleteness in knowledge graphs by predicting missing facts. This is particularly valuable for knowledge graphs in domains such as biology, medicine, and finance, where continuously evolving information may result in incomplete representations.

Ontology alignment

Graph learning can facilitate **ontology alignment** by capturing the structural and semantic relationships between entities in different ontologies. This is crucial for integrating information from diverse sources and ensuring interoperability between knowledge graphs.

For example, in a healthcare knowledge graph with multiple ontologies, graph learning can align the ontologies by recognizing relationships and similarities between entities, ensuring seamless integration of information from diverse medical sources.

Graph learning enriches the realm of knowledge graphs by providing sophisticated tools for representation learning, reasoning, and prediction. As the synergy between graph-based techniques and knowledge graphs continues to grow, we can anticipate increasingly robust and intelligent systems for organizing, querying, and extracting knowledge from complex, interconnected datasets.

Some other applications

Let's explore various other applications where graph learning is actively applied to address real-world problems.

Natural language processing

Natural language processing (**NLP**) encompasses several key applications where graph learning techniques can enhance language understanding and processing capabilities. Here are the main areas where graph-based approaches make significant contributions:

- **Semantic representation**: Capturing semantic relationships between words
- **Named entity recognition (NER)**: Enhancing entity recognition accuracy
- **Coreference resolution**: Improving resolution of references in text
- **Dependency parsing**: Accurate parsing of sentence structures
- **Sentiment analysis**: Nuanced sentiment analysis using graph connectivity
- **Question answering**: Retrieving answers based on semantic relationships
- **Dialogue systems**: Managing context in conversational **artificial intelligence** (**AI**) using graph structures
- **Graph-based language models**: Integrating contextual information for better language understanding

Cybersecurity

In cybersecurity, graph learning emerges as a formidable tool for analyzing complex networks of interconnected entities, such as devices and users. By leveraging graph-based models, cybersecurity experts can detect patterns, anomalies, and potential threats, fortifying digital defense mechanisms with a holistic and proactive approach.

The major applications of graph learning in cybersecurity include the following:

- **Anomaly detection**: Identifying unusual patterns in network traffic
- **Threat intelligence**: Integrating and analyzing threat intelligence data
- **Attack graph analysis**: Modeling and analyzing potential attack paths
- **User behavior analysis**: Detecting anomalous behavior based on user interactions
- **Vulnerability assessment**: Identifying and prioritizing system vulnerabilities
- **Fraud detection**: Uncovering fraudulent activities through graph patterns

Social networks

Graph learning helps with unveiling intricate patterns and relationships among users. By modeling social structures as graphs, this approach enables the extraction of valuable insights, facilitating several key applications:

- **Community detection**: Identifying cohesive groups in social network graphs
- **Influence prediction**: Predicting influence and information flow in networks
- **Recommendation systems**: Enhancing personalized recommendations with graph data
- **Fraud detection**: Uncovering fraudulent activities through social connections
- **Opinion dynamics**: Analyzing how opinions and information spread in a network
- **User engagement prediction**: Predicting user engagement based on social interactions

These use cases demonstrate the versatility of graph learning across different domains, showcasing its applicability in addressing diverse challenges, enabling a deeper understanding of social dynamics, and optimizing processes for building intelligence over ever-evolving interconnected data of the modern world.

Summary

Graphs offer a robust framework for modeling interconnected real-world systems, wherein *nodes* represent entities and *edges* capture relationships. Node-level learning is geared toward predicting the attributes and behaviors of individual nodes, facilitating applications such as personalized recommendations. On the other hand, edge-level learning delves into analyzing relationships between entities, supporting tasks such as link prediction and anomaly detection. Meanwhile, graph-level learning provides a holistic perspective to comprehend the overall structure, identify communities, and forecast emerging patterns, proving valuable in applications such as urban planning.

The real-world implementations of graph learning are evident in recommender systems, where it enhances capabilities such as neighbor-based suggestions, addresses implicit feedback, and tackles cold start problems. Additionally, knowledge graphs utilize graph learning techniques to generate entity and relationship embeddings, predict missing links, align ontologies, and complete missing information. Beyond recommender systems and knowledge graphs, graph learning extends its reach to diverse domains, including NLP, cybersecurity, social network analysis, and bioinformatics, to name a few.

In the next chapter, we'll delve further into the concept of graph representation learning, which aims to encode graph structures into low-dimensional vectors that are leveraged by various ML tasks.

3
Graph Representation Learning

Having explained why applying deep learning techniques to graph data is a worthy endeavor, let's jump right into the thick of things. In this chapter, we'll introduce you to **graph representation learning**.

First, we'll examine representation learning from the perspective of traditional (tabular data-based) **machine learning** (**ML**) and then extend the idea to the graph data space. Following this, we'll talk about the initial challenges that need to be addressed when you're trying to learn features within graph data. Next, you'll be introduced to a few simple graph representation learning algorithms, namely, **Node2Vec** and **DeepWalk**, and understand the differences between them. Finally, we'll discuss the limitations of such **shallow encoding** techniques and why we need algorithms with more firepower to capture more complex relationships in graphs.

We'll also introduce implementations of relevant algorithms in Python. We'll use Python as our language of choice and primarily use the **PyTorch Geometric** (**PyG**) library to implement our algorithms. Other libraries are popular as well (such as **Tensorflow-Graph Neural Networks (GNNs)**), but PyG seems to be the most established in the industry at the time of writing.

In this chapter, we'll cover the following topics:

- Representation learning – what is it?
- Graph representation learning
- A framework for graph learning
- DeepWalk
- Node2Vec
- Limitations of shallow encodings

Representation learning – what is it?

Modern ML-related tasks and experiments have settled into a standardized workflow pipeline. Here's a quick and simplified overview of the steps:

1. Convert the business/domain-specific problem into an ML problem (supervised or unsupervised, what metric is being optimized, baseline levels of metrics, and so on).
2. Get the data.
3. Pamper the data (by introducing new columns based on existing ones, imputing missing values, and more).
4. Train an ML model on the data and evaluate its performance on the test set. Iterate on this step with new models until a satisfactory performance is achieved.

One of the most important and time-consuming steps in this list is deciding how new columns can be created from the existing ones to add to the knowledge being specified in the data.

To understand this, let's understand the meaning of what a dataset is. A row in a dataset is effectively just a record of an event. The different columns (features) in a row represent the different variables (or dimensions, gauges, and so on), whose values were recorded at that event. Now, for ML models to learn useful information, the values of the primarily *useful* features must be recorded in the dataset. When we refer to features as useful, we mean those whose values, when changed, significantly alter the overall outcome of the event.

Let's try to understand this with an example. An extremely popular problem in the domain of **natural language processing** (**NLP**) is the task of predicting what the next word would be, given the previous words. We won't get into the nitty-gritty of this, instead just concentrating on a few scenarios where different feature choices have been made. The different features here are essentially the previous words:

```
Feature_1 (F1): The last word of the unfinished sentence.
Feature_2 (F2): The 2nd last word of the unfinished sentence.
Feature_3 (F3): The 3rd last word of the unfinished sentence.
… and so on.
```

Take the following unfinished sentence:

As the sun set over the horizon, the mountains cast elongated ___.

The features (along with the expected word, given a training dataset) would be as follows:

F11	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1	P
As	the	sun	set	over	the	horizon	the	mountains	cast	elongated	shadows

Table 3.1 – The features for predicting the next word

With this understanding of what features are, let's look at different subsets of features that can be taken as *eligible features* for model training. Let's take a look at three different cases:

```
Case 1: All features with a lookback window of 10.
{F1, F2, F3, …, F8, F9, F10}
Case 2: The 10th last word to the 5th last word
{F5, F6, F7, F8, F9, F10}
Case 3: All even positioned last words till the 10th position
{F2, F4, F6, F8, F10}
```

In this instance, we have the following training set (the words in parentheses indicate the word to be predicted by the model):

```
R1: As the sun set over the horizon, the sky turned a fiery (orange).
R2: As the sun set over the horizon, the clouds glowed with a golden
(hue).
R3: As the sun set over the horizon, the ocean shimmered with
reflected (light).
R4: As the sun set over the horizon, the landscape transformed into a
(silhouette).
```

Now, let's see what the dataset would look like once feature transformation has been performed for the three use cases.

Here's *Case 1*:

R	F10	F9	F8	F7	F6	F5	F4	F3	F2	F1
R1	sun	set	over	the	horizon	the	sky	turned	a	fiery
R2	set	over	the	horizon	the	clouds	glowed	with	a	golden
R3	sun	set	over	the	horizon	the	ocean	shimmed	with	reflected
R4	sun	set	over	the	horizon	the	landscape	transformed	into	a

Table 3.2 – Features with a lookback window of 10 (Case 1)

This is *Case 2*:

R	F10	F9	F8	F7	F6	F5
R1	sun	set	over	the	horizon	the
R2	set	over	the	horizon	the	clouds
R3	sun	set	over	the	horizon	the
R4	sun	set	over	the	horizon	the

Table 3.3 – Features with 10th-last to 5th-last window (Case 2)

Finally, we have *Case 3*:

R	F10	F8	F6	F4	F2
R1	sun	over	horizon	sky	a
R2	set	the	the	glowed	a
R3	sun	over	horizon	ocean	with
R4	sun	over	horizon	landscape	into

Table 3.4 – Features with even indexed words (Case 3)

In which case do you think the model would learn something useful?

Case 1, without a doubt. This example might be trivial, but it shows how decisions regarding feature selection heavily affect model performance down the pipeline. Feature selection is a step that involves taking the raw data and deciding on the useful truths within it; a mistake in this step can be devastating.

With this understanding as to why feature selection is an important step, let's go back to our discussion on representation learning. Representation learning is encountered in almost all subdomains of ML.

In the domain of images, representation learning was initially explored along the lines of traditional statistical approaches such as **principal component analysis** (**PCA**). However, given that the data to be learned from was in the form of pixel data, or equivalently, a 2D matrix of float values, other ideas that were more optimized for the task were explored. The most successful idea in this domain was the idea of using **convolutional filters** to extract meaningful patterns from the data matrix. This was the fundamental driving force behind most of the ML tasks that were conducted on images in the modern day. The **convolutional neural network** (**CNN**) has its initial layers reliant on finding the filters with the right values so that meaningful patterns can be extracted from the image when they're filtered through it.

In the domain of natural language, the key breakthrough in feature learning was understanding the dependence of every token on its previous tokens. A representation needed to be formulated that learned as we passed each token while maintaining a state of the learned knowledge thus far. **Recurrent neural networks** (**RNNs**) maintain the concept of memory where, as the tokens are passed sequentially, the memory vector gets updated based on the memory state and the new token. Other architectures, such as **long short-term memory** (**LSTM**), improve upon the RNN model to support longer-range dependencies and interactions between tokens. Other traditional methods have included algorithms such as **GloVe** and **Word2Vec**, which are of the shallow variant.

In tabular data, there are several tried and tested approaches to feature manipulation that can be used to denoise and extract meaningful information from the data. Common dimensionality reduction approaches such as PCA and **encoder-decoder networks** have proven effective in the industry. Using matrix factorization methods over incomplete and sparse interaction matrices to generate embeddings has been a very effective first step in building business-facing technologies such as search and recommendation engines.

The list of innovations in the field of representation learning is endless. The key takeaway is that representation learning is strongly tied to the domain of the problem being tackled. Effectively, it's a way to guide the algorithm so that it uses more effective techniques instead of a generic architecture, and it doesn't exploit the invariant patterns of the underlying data it's trying to learn from.

Graph representation learning

In the previous section, we talked about the need to perform representation learning on different types of data, such as images, tabular, and text. In this section, we'll try to extend this idea to graph data. Graph data, theoretically, is more expressive than all the other data representation methods we've dealt with so far (such as matrices for images, word tokens, and tables). With this expressivity comes the added challenge of finding a representation framework that captures relevant information, even though fewer constraints are enforced in the data representation itself. Words in text are sequential, pixels in images are represented as 2D matrices, and tabular data assumes independence of rows (most of the time).

Such inherent patterns in data allow us to exploit it during the representation learning step (think skip-grams for words and convolutional filters for images). However, the constraints in graphs are very loose – so loose in fact that there is no obvious pattern to be exploited. There are two primary challenges that graph data poses compared to other forms of data:

- **Increased computational costs**: One of the goals of representation learning is to output features that make the patterns more obvious for the model to learn from. As discussed in *Chapter 1*, there are several useful properties in graphs that can be exploited to make quick inferences. Understanding the shortest path between two nodes can be a useful feature to add to the representation, but computing that takes at least $O(|V|^2)$ time on average. Pattern mining using traditional techniques is a difficult process to undertake when we try to apply ML to graphs.
- **It's difficult to attack the problem in a distributed manner**: Modern ML pipelines are performant because of their ability to horizontally scale as the volume of data grows. Traditional data representations of images, texts, and tabular data can be easily parallelized owing to their inherent independence within data elements. Graph data is difficult to break down into smaller chunks, where one chunk has no dependence on another (essentially, this is graph partitioning). This is the biggest bottleneck and is a point of active research for making graph-based ML solutions feasible for production use cases. One great resource on this is *Graph neural networks meet with distributed graph partitioners and reconciliations* by Mu et al. (2023, `https://www.sciencedirect.com/science/article/abs/pii/S0925231222011894`).

Given these issues, it was quickly understood that traditional representation learning approaches that worked on other kinds of data wouldn't be useful for graph data. There needed to be a fresh approach tailored toward graph data. Even though the differences that we've mentioned concerning the problems in representation learning in graph data versus traditional data exist, graph representation learning maintains a partially common objective regarding representation learning in other domains. The objective of graph representation learning includes finding a representation mechanism that will reduce noise in the truth data, as well as highlight patterns that exist in the graph. The representation space also needs to be friendly toward modern-day ML algorithms so that gradients and scalar products can be easily computed.

Keeping all these objectives and constraints in mind, graph representation learning tries to find a transformation so that the following two goals can be achieved:

- **The representations capture the relationship between the nodes**. If two nodes are connected by an edge, the distance between the representations of these two nodes should be small. This concept should also hold to higher orders. So, if a subgraph is densely connected, the representations of the nodes of that subgraph should also form a dense cluster in the representation space.
- **The representations are optimized so that they solve the inference problem being tackled**. Popular graph problems include node classification, link prediction, and important node identification. Without this goal, the graph representation would remain too generic to solve interesting problems that lead to a business lift.

In conclusion, graph representation learning involves finding embeddings (or vectors) for every node of the graph so that vital information about the graph structure and information relevant to the inference problem can be captured in the embedding space. The first goal is often called the reconstruction constraint in this domain, whereby you can (to a certain degree) reconstruct the graph data, given the node embeddings. Before we delve deeper into various graph representation learning approaches, we must clarify two things:

- Graph data is rarely just a set of *nodes* (V) and *edges* (E). Often, all the nodes (and possibly the edges) contain side information, meaning additional fields are used to describe the nature of the nodes and edges. Modern representation techniques must also have a way to incorporate this information in the embeddings since they're often pivotal in the inference task.
- Graph representation learning is a ubiquitous step in graph learning. As with traditional ML approaches, some approaches involve distinct steps of learning the representation embeddings, followed by learning the model that's being used for inference. Other approaches combine both steps, where the embedding step is handled internally within the model that's responsible for outputting the inference output. In the world of graph learning, the same two approaches apply.

Now, let's dive into some graph representation learning approaches. There has been substantial work on graph representation learning, and different approaches have tried to tackle different aspects of the problem. Without diving too deep into the chronological development of algorithms in this field, we'll discuss the most relevant approaches in the industry today. All approaches can be segregated into two concepts: **shallow encodings** and **deep encodings**.

To explain the difference in a single statement, encodings are *shallow* when only the fields of the node embeddings need to be estimated; if more parameters need to be estimated alongside the fields of the node embeddings, then the encoding method is called *deep* encoding.

The remainder of this chapter will primarily focus on the most popular shallow encoding algorithms in graph representation learning: DeepWalk and Node2Vec. The different GNN architectures that will be mentioned in the following chapters are examples of deep encoding techniques.

A framework for graph learning

If we take a holistic view of the approaches that are followed for learning inference models on graphs, we'll notice a pattern. Every solution can be divided into three distinct steps:

1. The first step involves coming up with a mechanism to find a **local subgraph**, given a node in the graph. This term needs to be defined here. For example, the graph containing all the nodes that are directly connected to an edge of the concerned node can be a local subgraph. Another example can be the set of nodes that have a first or second-degree connection to the concerned node. This local subgraph is often called the receptive field of the concerned node in academic literature.
2. The second step involves a mechanism that takes input from the concerned node and its receptive field and outputs the node embedding. The node embedding is simply a vector of real values of a certain dimension. It's important to have a similarity metric defined in this metric space. A low similarity score between two vectors suggests that they are close to each other in this space.
3. The final step involves defining a learning objective. A learning objective is a function that tries to emulate what the learned node embeddings need to be optimized for. This can involve trying to mimic the graph structure, where if nodes are connected by edges, their embeddings will be more similar to each other. Alternatively, it could involve some graph inference tasks, such as optimizing the embeddings so that the nodes can be classified correctly.

This framework should always be in the back of your mind while you go through the rest of this book. Several algorithms and learning models will be outlined in the following few chapters, and you should always be looking to answer the question of how different components of the algorithm conform to this framework. With this knowledge at hand, let's look at the DeepWalk algorithm and its varieties.

DeepWalk

DeepWalk is a learning technique that falls under the subcategory of algorithms where a random walk must be performed over the graph to find optimal embeddings. To make sense of random walk-based learning techniques, let's rewind and pick up from where the previous section left off. Remember that the objective of this exercise is to come up with an embedding for every node in the graph so that pairs of embeddings are very similar in the vector space if – and only if – the nodes these embeddings represent are also very similar in the graph.

To achieve this goal, we must define what *similar* means in both the vector space and the graph. Similarity in the vector space is often defined using the cosine similarity function (other similarity functions can also be used, such as L1-similarity, but for the graph use case, cosine similarity remains the most popular). Let's start by defining cosine similarity.

Let's say we have the embeddings of two nodes, n_1 and n_2:

$$v_{n_1} = \left(x_1, x_2, x_3, \ldots, x_d\right)$$
$$v_{n_2} = \left(y_1, y_2, y_3, \ldots, y_d\right)$$

In this case, the cosine similarity function, $Cos\left(v_{n_1}, v_{n_2}\right)$, is defined as follows:

$$\cos\left(v_{n_1}, v_{n_2}\right) = \frac{\left(x_1 . y_1 + x_2 . y_2 + x_3 . y_3 + \ldots + x_d . y_d\right)}{\sqrt{\left(x_1^2 + x_2^2 + \ldots + x_d^2\right)\left(y_1^2 + y_2^2 + \ldots + y_d^2\right)}}$$

The interpretation is well-studied in the field of ML, so not much has been elaborated here. In short, you can think of cosine similarity as the score of how similarly oriented the two vectors are in the vector space. Vectors that point almost in a similar direction will have a cosine similarity score closer to 1, while two vectors that are perpendicular to each other will have a score closer to 0.

The easy part was defining the similarity score between the embeddings. Now, we need to come up with a similarity score between the nodes of the graph.

Random walk – the what and the why

How do we define the similarity between two nodes in a graph? Could it be as simple as saying, assign a score of 1 if they are directly connected by an edge; if they are not directly connected but share common neighbors, assign a score equal to the natural logarithm of the number of common neighbors; if neither condition is met, assign a score of 0.

A lot of thought can be put into finding the best definition of similarity between the two nodes, and you can imagine how this definition would need to approximate a key structural property of the graph. All heuristics that would try to define the similarity score between the nodes would have advantages and disadvantages. However, there's one idea we can implement that's simple and mathematically elegant: **random walks**.

The idea of random walks is quite self-explanatory. This algorithm is used to find a neighborhood for some node in the graph, say *v*. Based on some strategy, say *R* (we'll explain what strategies mean in a bit), we try to find the elements that would be part of the neighborhood of *v*; here, the neighborhood is $N_R\left(v\right)$. The following steps explain the algorithm:

1. Starting from *v*, we ask the strategy, *R*, to decide (with some degree of randomness) which node that's connected to *v* we should jump to. Let that node be called n_1. Add n_1 to the set, $N_R\left(v\right)$.

2. Repeat *Step 1*, but start from n_1 instead of v. The output of the strategy would be n_2, and that would be appended to $N_R(v)$. Repeat this for a fixed number of steps until you have enough entries in your set.

That's it! We're pretty much done with the random walk step of the algorithm. Before we consider how random walks generate neighborhoods, let's discuss strategies. The strategy, R, decides which node should we jump to from the previous node. The simplest strategy is a random choice where, given all the connected nodes, you choose a node at random with the same probability as all others. Other strategies can be employed as well, where, for example, the unvisited nodes can be given more bias, instead of us adding the same node to the neighborhood repeatedly. The different choice of strategies often ends up being the differentiating factor between different shallow embedding learning algorithms in this class.

Why random walks? Random walks are a good way to sample the important features of the graph efficiently. First, if we notice that a node occurs within the random walk neighborhood of another node with high probability, we can probably conclude they're supposed to be very similar to each other. Such a technique doesn't rely on hacky heuristics, which are often limited by the degree of connections. Random walk-based sampling doesn't need to worry about the degree of connections to figure out the best candidates to be part of the neighborhood at a statistical level. Second, it's a quick and efficient way to sample. The training step doesn't need to evaluate across all the nodes in the graph; it just needs to concern itself with the ones in the neighborhood.

In the next subsection, we'll build on our understanding of random walks and use it to come up with a method of estimating the embedding components.

Estimating the node embeddings

Now that we've learned the neighborhood set $N_R(v)$ for the node, v, let's understand how it's relevant to the task of learning node embeddings. Recall that before we started talking about random walks, our subproblem of concern was to find a way to estimate the similarity of two nodes in a graph. Coming up with a similarity function over the nodes of a graph is a tough ask, but can we make some guided assumptions about this similarity score?

An important assumption could be as follows: "*When two nodes are similar, one node likely lies within the neighborhood of the other.*" If we assume this statement to be true, we've pretty much solved our problem. Now, we can leverage this assumption to come up with the node embeddings. Without delving deep into probability magic (such as that of likelihood functions, and so on), the crux of the idea is that if this assumption is true in the graph space, then it must also be true in the vector space where the embeddings are defined. If that's the case, we need to find the embedding fields so that the likelihood function is maximized whenever one node is within the neighborhood of the other.

So, if z_v is an embedding vector of node v, we can find the components of z_v so that the following value is being maximized:

$$L_v = \sum_{u \in N_R(v)} -\left(\log\left(P\left(u \middle| z_v\right)\right)\right)$$

The negative log of the probability is a form of the log-likelihood function. Now, since this must be true across all the nodes and their embeddings, we must optimize the components while keeping all the nodes in mind, essentially maximizing:

$$L = \sum_{v \in V} L_v = \sum_{v \in V} \sum_{u \in N_R(v)} -\left(\log\left(P\left(u \middle| z_v\right)\right)\right)$$

The final step is to tie the probability function to the similarity function we defined previously. A common way of creating a probability distribution within an embedding space is to use a `softmax` function, which converts a function into a probability density function:

$$P\left(z_u \middle| z_v\right) = softmax\left(sim(z_u, z_v)\right) = \frac{\exp\left(sim(z_u, z_v)\right)}{\sum_{u \in V} \exp\left(sim(z_u, z_v)\right)}$$

Here, $sim(z_u, z_v)$ is simply the cosine similarity function we defined previously. By plugging this definition back into our optimization metric, we get the final parameterized form of the loss function that needs to be optimized:

$$L = \sum_{v \in V} \sum_{u \in N_R(v)} -\left(\log\left(\frac{\exp\left(sim(z_u, z_v)\right)}{\sum_{u \in V} \exp\left(sim(z_u, z_v)\right)}\right)\right)$$

Using gradient descent, we'll find the embeddings, z, so that L is maximized. This will ensure we find the embeddings that satisfy our criterion.

Note that there are a few other steps in the optimization process that make the process computationally feasible, with one of the most important being the concept of negative sampling. Here, instead of calculating the normalization component (the denominator) of the `softmax` function across all nodes on each iteration, we take a few random nodes that aren't in the neighborhood of the concerned node and calculate the sum over that. This type of optimization problem is called **noise contrastive estimations**, and it's a popular technique in NLP learning tasks. It's often called the **skip-gram model**.

As we conclude this section, it might be a good time to mention that the preceding algorithm in its entirety is termed *DeepWalk*, as stated in the original paper, which was published in 2014 (`https://dl.acm.org/doi/10.1145/2623330.2623732`). The DeepWalk algorithm is an efficient process of estimating shallow encodings. However, the simplicity of the approach is one of its major shortcomings: the random unbiased nature of the random walk strategy often wanders too far away from the concerned node, so it samples neighborhoods that aren't very local to the node. As a result, the embeddings aren't optimized based on the most local information of the node. Several other algorithms build on the work that was done in this paper. We'll talk about one such prominent improvement in the next section, known as *Node2Vec*.

Here's the pseudocode for DeepWalk:

```
function DeepWalk(Graph G, walk_length L, num_walks R, dimensions d):
    Initialize walks = []

    for each node v in G:
        for i = 1 to R:
            walk = RandomWalk(G, v, L)
            append walk to walks

    model = Word2Vec(walks, dimensions=d)
    return model

function RandomWalk(Graph G, start_node v, length L):
    walk = [v]
    for i = 1 to L:
        neighbors = GetNeighbors(G, v)
        next_node = RandomChoice(neighbors)
        append next_node to walk
        v = next_node
    return walk
```

The preceding pseudocode outlines two main functions:

- First, `DeepWalk` generates multiple random walks for each node in the graph and uses Word2Vec to create embeddings
- Second, `RandomWalk` performs a single random walk, starting from a given node for a specified length

Node2Vec

The DeepWalk algorithm uses unbiased randomized walks to generate the neighborhood of any concerned node. Its unbiased nature ensures the graph structure is captured in the best possible manner statistically, but, in practice, this is often the less optimal choice. The premise of Node2Vec is that we introduce bias in the random walk strategy to ensure that sampling is done in such a way that both the local and global structures of the graph are represented in the neighborhood. Most of the other concepts in Node2Vec are the same as those for DeepWalk, including the learning objective and the optimization step.

Before we delve into the nitty-gritty of the algorithm, let's do a quick recap of graph traversal approaches.

Graph traversal approaches

As we covered briefly in *Chapter 1*, the two most popular graph traversal approaches are **breadth-first search** (**BFS**) and **depth-first search** (**DFS**). BFS is the local first approach to graph exploration where, given a starting node, all first-degree connections are explored before we venture away from the starting node. DFS, on the other hand, takes a global first approach to graph exploration, where the impetus is to explore as deep into the graph as possible before backtracking upon reaching a leaf node. The random walk strategy that's employed in the DeepWalk algorithm is statistically closer to the DFS approach than the BFS approach, which is why the local structure is under-represented in the neighborhood.

How can a random walk strategy emulate BFS and DFS?

First, let's consider the case for DFS. In a random walk process, when the current position is on some node, we have two choices: visit the last node that was visited or visit some other node. If we can ensure that the first option happens only when the current node has no other connected nodes, then we can ensure the random walk follows DFS traversal.

For a random walk to emulate BFS, a bit more consideration is needed. First, the random walk entity needs to keep track of which node it came from in the last step. Now, from the current node, we have three options: go back to the previous node, go to a node that's further away from the previous node, or go to a node that's equidistant from the previous node as the current node is. If we minimize the chances of the second option happening, we're effectively left with a BFS traversal.

So, we can add biases to the random walk algorithm to emulate the BFS and DFS traversal patterns. With this knowledge, we can enforce finer control over the random walk so that the neighborhoods contain local structure and global structure representations with the ratio we're interested in.

Finalizing the random walk strategy

Let's formalize the strategy mentioned previously. For this, we'll use two hyperparameters: `p` and `q`. The first hyperparameter, `p`, is related to a weight that decides how likely the random walk will go back to the node it came from in the last step. The second hyperparameter, `q`, decides how likely the walk will venture off to a node that's further away from the previous node or not. It can also be interpreted as a parameter that decides how much preference the BFS strategy gets over the DFS strategy. The example in *Figure 3.1* can clarify this:

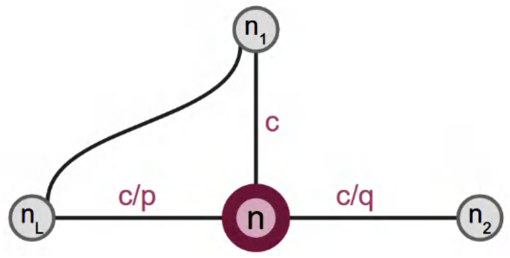


Figure 3.1 – A small graph showing the effect of the Node2Vec hyperparameters

Take a look at this graph. Here, the random walk has migrated from node n_L to n in the last step. In the current step, it needs to decide which node it should migrate to, with its options being n_1, n_2, or n_L. The probabilities of the next migration are decided by the hyperparameters, `p` and `q`. Let `c` be the probability that the next migration is to n_1. Then, the probability of migration to n_L is `c/p`, while the probability of migration to n_2 is `c/q`. Here, `c` should be such that the sum of all probabilities adds up to 1.

To clarify this, the values of the probabilities are the way they are because of what each node represents; n_L is the last visited node, which is why its visit is weighed additionally by the `p` value. Here, n_1 is the

BFS option since it is at the same distance from n_L as n is, while n_2 is the DFS option since it is further away from n_L. This is why the ratio of their visits is q.

With this strategy of assigning biases to each step of neighborhood creation, we can ensure the neighborhood consists of representatives from both the local and global context of the concerned node. Note that this random walk strategy is called a random walk strategy of the second order since we need to maintain a state – that is, the knowledge of the previous state from where the walk has migrated.

Here's the pseudocode for Node2Vec:

```
function Node2Vec(Graph G, walk_length L, num_walks R, dimensions d,
p, q):
    Initialize walks = []

    for each node v in G:
        for i = 1 to R:
            walk = BiasedRandomWalk(G, v, L, p, q)
            append walk to walks

    model = Word2Vec(walks, dimensions=d)
    return model

function BiasedRandomWalk(Graph G, start_node v, length L, p, q):
    walk = [v]
    for i = 1 to L:
        current = walk[-1]
        previous = walk[-2] if len(walk) > 1 else None
        next_node = SampleNextNode(G, current, previous, p, q)
        append next_node to walk
    return walk
```

Node2Vec extends DeepWalk by introducing biased random walks that are controlled by the `p` and `q` parameters. The `BiasedRandomWalk` function uses these parameters to balance between exploring local neighborhoods (`q`) and reaching farther nodes (`p`), allowing for a more flexible exploration of the graph structure.

Node2Vec versus DeepWalk

The steps that follow are the same as the ones that were mentioned for DeepWalk. We try to maximize the likelihood that the embeddings within the neighborhood of the concerned node are most similar to the concerned node. This optimization step, when performed across all the nodes, gives us the optimal embeddings. The difference from DeepWalk is in terms of what the embeddings are being optimized for. In DeepWalk, the choice of neighbors for a node was different than it was for the Node2Vec scenario.

With that, we've covered the two most popular shallow graph representation learning algorithms. We've learned how DeepWalk and Node2Vec, two similar algorithms, can elegantly employ the random walk approach to generate shallow node embeddings. However, we need to understand the limitations of such approaches since such restrictions will act as motivation for the topics that will be discussed later in this book to be used.

Limitations of shallow encodings

Shallow embeddings have the advantage of being easy to understand and relatively easy to implement. However, they have several disadvantages, especially compared to deep encoding techniques:

- **This method can't incorporate node or link-level features**. When using graph data, it's common to have auxiliary information attached to every node or every edge, to describe further properties. By default, the random walk approaches aren't capable of incorporating such information in their embeddings.
- **They can be computationally expensive**. How? If we're interested in embeddings of some dimension, `d`, and the number of nodes is `v`, then we need to learn a total of `v.d` values. Deep approaches with hidden layers would likely have much lower parameters to learn, making the process more computationally efficient.
- Building from the previous point, since there are so many parameters to learn in this approach, **we often can't utilize the advantages that come with making the representations denser**. Denser representations (which use fewer parameters) can often reduce the amount of noise that's learned. Obviously, below a minimum threshold, the number of parameters would become too low to be able to effectively represent the complexity of the graph data being learned.
- Finally, **the learning approaches for shallow encodings provide no provision to incorporate the graph inference tasks within the learning problem**. The embeddings are learned based on graph structure and are to be used for generic purposes, instead being optimized for one inference task.

Many of these limitations can be overcome with more sophisticated architectures that are optimized to learn node embeddings.

Summary

In this chapter, we introduced graph representation learning, a fundamental concept in the domain of using ML on graph data. First, we discussed what representation learning is, in the general sense in ML. When concentrating solely on graphs, you learned that the primary objective of representation learning is to find embeddings that can emulate the structure of the graph, as well as learn important concepts that are necessary for the inference task, if any.

We also explored DeepWalk and Node2Vec, two popular graph representation learning approaches, which comprise a class of algorithms that use random walks to generate a neighborhood for a node. Based on this neighborhood, you can optimize the embedding values so that the embeddings of the nodes in the neighborhood are highly similar to those of the embeddings of the concerned node. Finally, we looked at the drawbacks of using these approaches in practice.

In the next chapter, we'll concentrate on the most popular deep learning architectures that can be used to learn graph node embeddings. You'll learn how such architectures exploit patterns in graphs while maintaining the invariants in the graph data.

Part 2: Advanced Graph Learning Techniques

In this part of the book, you will explore advanced concepts in graph learning, including deep learning architectures for graphs, common challenges in the field, and the integration of large language models. You will learn about state-of-the-art approaches, technical challenges, and emerging solutions in graph-based AI systems.

This part has the following chapters:

4
Deep Learning Models for Graphs

In recent years, the field of machine learning has witnessed a paradigm shift with the emergence of **graph neural networks** (**GNNs**) as powerful tools for addressing prediction tasks on graph-structured data. Here, we'll delve into the transformative potential of GNNs, highlighting their role as optimizable transformations capable of handling diverse graph attributes, such as nodes, edges, and global context while preserving crucial graph symmetries, particularly permutation invariances.

The foundation of GNNs lies in the **message-passing neural network** (**MPNN**) framework. Through this framework, GNNs leverage a sophisticated mechanism for information exchange and aggregation across graph structures, enabling the model to capture intricate relationships and dependencies within the data.

One distinctive feature of GNNs is their adherence to a *graph-in, graph-out* architecture. This means that the model accepts a graph as input, equipped with information embedded in its nodes, edges, and global context. This inherent structure aligns with many real-world problems where data exhibits complex relationships and dependencies best represented as graphs.

GNNs excel in their ability to perform a progressive embedding transformation on the input graph without altering its connectivity. This progressive transformation ensures that the model refines its understanding of the underlying patterns and structures within the data, contributing to enhanced predictive capabilities.

We'll cover the following topics in this chapter:

- Message passing in graphs
- Decoding GNNs
- **Graph convolutional networks** (**GCNs**)
- **Graph Sample and Aggregation** (**GraphSAGE**)
- **Graph attention networks** (**GATs**)

Technical requirements

This chapter requires a basic understanding of graphs and representation learning as covered in previous chapters. The code present in the chapter and on GitHub can be used directly on Google Colab with an additional installation of the **PyTorch Geometric** (**PyG**) package. The code examples for the book are available in its GitHub repository: `https://github.com/PacktPublishing/Applied-Deep-Learning-on-Graphs`.

Message passing in graphs

Unlike traditional neural networks, GNNs need to account for the inherent structure of the graph, allowing nodes to exchange information and update their representations based on their local neighborhoods. This core mechanism is achieved through **message passing**, a process of iteratively passing messages between nodes and aggregating information from their neighbors.

GNNs operate on graph-structured data and use a message-passing mechanism to update node representations based on information from neighboring nodes. Let's delve into the mathematical explanation of message passing in GNNs.

Consider an undirected graph $G = (V, E)$, where V is the set of nodes and E is the set of edges. Each node v in V has an associated feature vector x_v. The goal of a GNN is to learn a representation h_v for each node v that captures information from its neighborhood.

The basic message-passing operation in a GNN can be broken down into a series of steps:

1. **Aggregation of messages**:

    I. For each node v, gather information from its neighbors.

    II. Let $N(v)$ represent the set of neighbors of node v.

    The aggregated message m_v for node v is computed by aggregating information from its neighbors:

    $$m_v = aggregate\left(\left\{h_u : u \in N(v)\right\}\right)$$

    The aggregation function can vary (e.g., sum, mean, or attention-weighted sum).

1. **UPDATE function**:

    I. Update the node representation h_v based on the aggregated message m_v and the current node representation h_v.

    II. The `UPDATE` function *update* $(\cdot)$ is a neural network layer that takes the aggregated message and the current node representation as input and produces the updated representation:

    $$h'_v = update\left(h_v, m_v\right)$$

    The `UPDATE` function typically involves a neural network layer with learnable parameters.

These steps are iteratively applied for a fixed number of times or until convergence to refine the node representations. The overall process can be expressed in a few equations:

$$m_v = aggregate\left(h_u : u \in N(v)\right)$$

$$h'_v = update\left(h_v, m_v\right)$$

These equations capture the essence of the message-passing mechanism in a GNN. The specific choices for the aggregation function, `UPDATE` function, and the number of iterations depend on the architecture of the GNN (e.g., GraphSAGE, GCN, **gated graph neural networks** (**GGNNs**), etc.).

For instance, in a simple `GraphSAGE` formulation, the aggregation function might be a mean operation, and the `UPDATE` function might be a simple neural network layer:

$$m_v = \frac{1}{|N(v)|} \sum_{u \in N(v)} h_u$$

$$h'_v = \sigma\left(W \cdot concat\left(h_v, m_v\right)\right)$$

Here, σ is an activation function, W is a learnable weight matrix, and $concat(\cdot)$ is the concatenation operation.

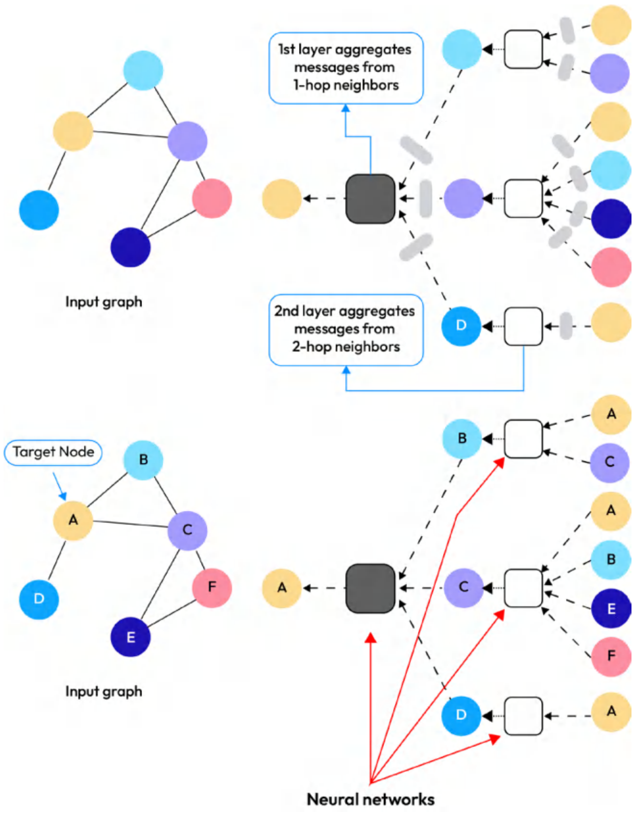

1st layer aggregates messages from 1-hop neighbors
2nd layer aggregates messages from 2-hop neighbors
Input graph
Target Node
A
B
C
D
E
F
Input graph
Neural networks

Figure 4.1 – Message passing in graphs

Through these iterative steps, message passing enables nodes to learn representations that capture not only their own intrinsic features but also the information from their connected neighbors and the overall structure of the graph. This allows GNNs to effectively model complex relationships and dependencies within graph-structured data.

Now, let's try to understand how to formally define a GNN.

Decoding GNNs

A **GNN** is a neural network architecture designed to operate on graph-structured data. It learns a function that maps a graph and its associated features to a set of node-level, edge-level, or graph-level outputs. The following is a formal mathematical definition of a GNN.

Given a graph $G = (V, E)$, where V is the set of nodes and E is the set of edges, let $X \in \mathbb{R}^{|V| \times d}$ be the node feature matrix, where each row $x_v \in \mathbb{R}^d$represents the features of node $v \in V$.

A GNN is a function $f_\theta : G \times \mathbb{R}^{|V| \times d} \rightarrow \mathbb{R}^{|V| \times d'}$ parameterized by learnable weights θ, which maps the graph G and its node features X to a new set of node representations $H \in \mathbb{R}^{|V| \times d'}$, where d' is the dimensionality of the output node representations.

The function f_θ is computed through a series of message passing and aggregation steps, typically organized into L layers. At each layer $l \in \{1, \ldots, L\}$, the node representations are updated as follows:

$$h_v^l = UPDATE^{(l)}\left(h_v^{l-1}, AGG^{(l)}\left(\left\{h_u^{l-1} : u \in N(v)\right\}\right)\right)$$

Let's break this down:

- $h_v^{(l)} \epsilon\, \mathbb{R}^{d_l}$ is the representation of node v at layer l, with $h_v^0 = x_v$. Here, $\mathbb{R}^{(d_l)}$ represents a real-valued vector space of dimension d_l.
- The **UPDATE function** $UPDATE^l : \mathbb{R}^{(d_{l-1})} \mathrm{X}\, \mathbb{R}^{(d_l)} \rightarrow \mathbb{R}^{(d_l)}$ is a learnable function that updates the node representation based on its previous representation and the aggregated messages from its neighbors.
- The **aggregation (AGG) function** $AGG^{(l)}:\mathbb{R}^{d_{l-1}{}^*} \rightarrow \mathbb{R}^{d_l}$ is a permutation-invariant aggregation function that combines the representations of the neighboring nodes. Common choices include sum, mean, and max.
- $N(v)$ denotes the set of neighbors of node v in the graph G.

After L layers of message passing and aggregation, the final node representations are given by $H = h_v^L$ for all $v \in V$.

The `UPDATE` function is typically implemented as neural networks, such as **multi-layer perceptrons** (**MLPs**) or attention mechanisms, with learnable parameters.

For graph-level tasks, a **READOUT function** is applied to the final node representations to obtain a graph-level representation:

$$h_G = READOUT\left(\left\{h_v^L : v \in V\right\}\right)$$

Here, `READOUT` is a permutation-invariant function that aggregates the node representations into a single vector, such as sum, mean, or a more complex pooling operation.

The graph-level representation h_G can then be used for downstream tasks, such as graph classification or regression.

This is a general formulation of GNNs, and there are many specific architectures that fall under this framework, such as GCNs, GraphSAGE, GATs, and MPNNs, each with their own variations of the `UPDATE`, `AGG`, and `READOUT` functions.

Let's understand how graph learning borrows the concept of convolution networks and leverages it to extract learnings from a graph.

GCNs

GCNs are a specific type of GNN that extend the concept of convolution to graph-structured data. GCNs learn node representations by aggregating information from neighboring nodes, allowing for the capture of both node features and graph structure.

In a GCN, the graph convolution operation at layer l is defined as follows:

$$H^{(l+1)} = \sigma(D^{-\frac{1}{2}} A\, D^{\frac{1}{2}}\, H^{(l)}\, W^{(l)})$$

Let's break this down:

- $H^{(l)} \in R^{|V| \times d_l}$ is the matrix of node representations at layer l, with $H(0) = X$ (input node features).
- $\widehat{A} = A + I$ is the adjacency matrix A with added self-loops, where I is the identity matrix.
- $\widehat{D}$ is the diagonal degree matrix of $\widehat{A}$, with $\widehat{D_{ii}} = \sum_j A_{ij}$.
- $W^{(l)} \in R^{d_l\, x\, d_{l+1}}$ is a learnable weight matrix for layer l.
- $\sigma(\cdot)$ is a non-linear activation function, such as the **rectified linear unit** (**ReLU**) function or sigmoid function.

The term $D^{-\frac{1}{2}} A\, D^{\frac{1}{2}}$ is the symmetrically normalized adjacency matrix, which ensures that the scale of the node representations remains consistent across layers.

Imagine a citation network, where each node stands for a scientific paper and each edge represents a citation connecting two papers. Each paper has a feature vector representing its content (e.g., bag-of-words). A GCN can be used to classify the papers into different categories (e.g., computer science, physics, biology) by learning node representations that capture both the content and the citation structure of the network.

Building on our mathematical understanding of the GCN, let's look at a piece of sample code leveraging PyG:

```
import torch
import torch.nn.functional as F
from torch_geometric.datasets import Planetoid
from torch_geometric.nn import GCNConv

# Load the Cora dataset
dataset = Planetoid(root='data/Cora', name='Cora')
```

For this example, we import the necessary libraries and load the **Cora dataset** using the `Planetoid` class from PyG. The Cora dataset is a citation network dataset, where *nodes* represent scientific papers and *edges* represent citations between papers. The dataset contains 2,708 nodes, 10,556 edges, and 7 classes representing different research areas:

```
# Get the graph data
data = dataset[0]

# Print some statistics about the graph
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Number of features: {data.num_features}')
print(f'Number of classes: {dataset.num_classes}')
```

Here, we access the graph data using `dataset[0]`. We then print some statistics about the graph, including the number of nodes, edges, features, and classes:

```
Number of nodes: 2708
Number of edges: 10556
Number of features: 1433
Number of classes: 7
```

Now that we understand what the data looks like, let's put down the building blocks of the model:

```
# Define the GCN model
class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GCN, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)

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

Next, we define the GCN model. The model consists of two GCN layers (`GCNConv`) with a hidden layer in between. The `__init__` method initializes the layers with the specified input, hidden, and output dimensions. The `forward` method defines the forward pass of the model, where `x` and `edge_index` are passed through the GCN layers. ReLU activation and dropout are applied after the first layer, and `log-softmax` is applied to the output of the second layer:

```
# Set the model parameters
in_channels = dataset.num_node_features
hidden_channels = 16
out_channels = dataset.num_classes

# Create an instance of the GCN model
model = GCN(in_channels, hidden_channels, out_channels)
```

Here, we set the model parameters based on the dataset. The input dimension (`in_channels`) is set to the number of node features in the dataset, the hidden dimension (`hidden_channels`) is set to `16`, and the output dimension (`out_channels`) is set to the number of classes in the dataset. We then create an instance of the GCN model with these parameters:

```
# Define the optimizer and loss function
optimizer = torch.optim.Adam(
    model.parameters(), lr=0.01, weight_decay=5e-4)
criterion = torch.nn.NLLLoss()

# Train the model
model.train()
for epoch in range(200):
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = criterion(out[data.train_mask],
                     data.y[data.train_mask])
    loss.backward()
    optimizer.step()
```

In this part, we define the optimizer (**Adam**) and the loss function (**negative log-likelihood loss (NLLLoss)**) for training the model. We set the learning rate to `0.01` and the weight decay to `5e-4`.

We then train the model for 200 epochs. In each epoch, we do the following:

1. Zero the gradients of the optimizer.
2. Perform a forward pass of the model on the node features and edge index.

3. Compute the loss using the model's output and the ground truth labels for the training nodes (specified by `data.train_mask`).
4. Perform a backward pass to compute the gradients.
5. Update the model parameters using the optimizer:

```
# Evaluate the model
model.eval()
_, pred = model(data.x, data.edge_index).max(dim=1)
correct = float(pred[data.test_mask].eq(
    data.y[data.test_mask]).sum().item())
accuracy = correct / data.test_mask.sum().item()
print(f'Accuracy: {accuracy:.4f}')

Accuracy: 0.8000
```

Finally, we evaluate the trained model on the test set. We set the model to evaluation mode using `model.eval()`. We perform a forward pass on the entire graph and obtain the predicted class labels using `max(dim=1)`. We then compute the accuracy by comparing the predicted labels with the ground truth labels for the test nodes (specified by `data.test_mask`).

This code provides a basic implementation of GCN using PyG on the Cora dataset.

Note

Make sure you have PyG installed before running this code. You can install it using `pip install torch-geometric`.

Overall, this code creates an instance of the GCN model, trains it on the Cora dataset using the specified hyperparameters (hidden units, learning rate, epochs), and evaluates the trained model on the test set to measure its performance in terms of accuracy.

Using GCNs for different graph tasks

GCNs can be utilized to learn and perform tasks at different levels in a graph. The following can be performed with GCNs:

- **Node-level tasks**: GCNs can be used for **node classification**, where the goal is to predict the label of each node in the graph. This is demonstrated in the previous example, where the GCN is used to classify nodes in the Cora citation network.
- **Edge-level tasks**: GCNs can be adapted for **edge prediction** or **link prediction** tasks, where the goal is to predict the existence or attributes of edges in the graph. To do this, the node representations learned by the GCN can be used to compute edge scores or probabilities.

- **Graph-level tasks**: GCNs can be used for **graph classification** or **regression** tasks, where the goal is to predict a label or a continuous value for an entire graph. To achieve this, a pooling operation (e.g., global mean pooling or global max pooling) is applied to the node representations learned by the GCN to obtain a graph-level representation, which is then fed into a classifier or regressor.

GCNs are a powerful and widely used type of GNN that can effectively learn node representations by incorporating both node features and graph structure. They have shown strong performance on various graph-based tasks and can be adapted for node-level, edge-level, and graph-level problems.

Over time, many optimizations over vanilla GCNs have been proposed and utilized in the industry. One such optimization, especially for scaling the graph learning process, is GraphSAGE.

GraphSAGE

GraphSAGE introduces a scalable and adaptive approach to graph representation learning, addressing some limitations of GCN and enhancing scalability. At its core, GraphSAGE employs a neighborhood sampling and aggregation strategy, diverging from the fixed-weight aggregation mechanism of GCN.

In GraphSAGE, the process of learning node representations involves iteratively sampling and aggregating information from local neighborhoods. Let $G = (V, E)$ be a graph with nodes V and edges E, and f_i^l denote the embedding of node *i* at layer *l*. The update rule for GraphSAGE can be expressed as follows:

$$f_i^{l+1} = Aggregate\left(\left\{f_j^l, \forall j \in SampleNeighbors(i)\right\}\right)$$

Here, $SampleNeighbors(i)$ represents a dynamically sampled subset of neighbors for node *i* at each iteration. This adaptability allows GraphSAGE to scale more efficiently compared to GCN, especially in scenarios where the graph is large or when computational resources are limited, maintaining scalability.

The structure for PyG code remains the same as for the GCN; we will just be using the `GraphSAGE` module from `torch_geometric.nn`.

To modify the previous code to use GraphSAGE instead of GCN, you need to make a few changes. Here are the lines you need to update:

1. Replace the `import` statement for `GCNConv` with `SAGEConv`:

```
from torch_geometric.nn import SAGEConv
```

2. Update the `GCN` model class to use `SAGEConv` layers instead of `GCNConv`:

```
# Define the GraphSAGE model
class GraphSAGE(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels,
                 out_channels):
        super(GraphSAGE, self).__init__()
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, out_channels)

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

3. Update the `model` creation line to use the `GraphSAGE` model instead of `GCN`:

```
# Create an instance of the GraphSAGE model
model = GraphSAGE(in_channels, hidden_channels, out_channels)
```

With these changes, the code will now use the GraphSAGE model instead of GCN. The rest of the code, including loading the dataset, training, and evaluation, remains the same.

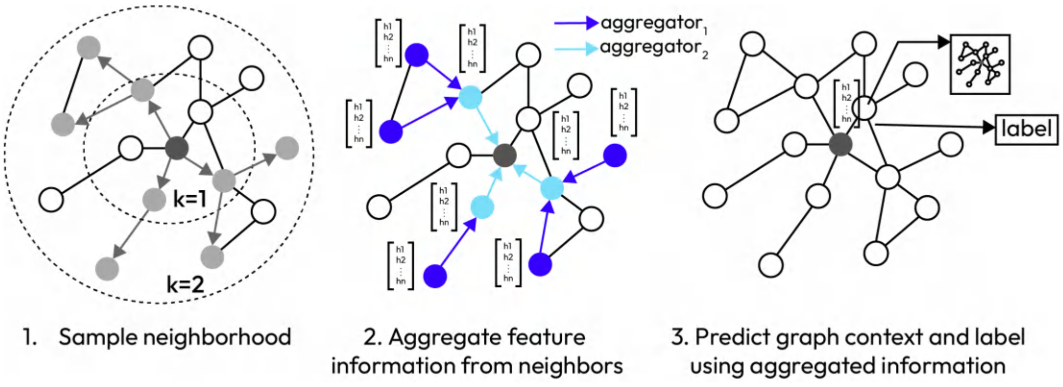

k=1
k=2
1. Sample neighborhood
aggregator$_1$
aggregator$_2$
2. Aggregate feature information from neighbors
label
3. Predict graph context and label using aggregated information

Figure 4.2 – GraphSAGE network leveraging neighborhood sampling

At times, it is a good idea to assign different weights to neighbors based on their relevance to a particular node. We will now look at GATs, which borrow the concept of attention from language models.

GATs

GATs are an extension of GCNs that incorporate an attention mechanism to assign different weights to neighboring nodes based on their relevance. While GCNs apply a fixed aggregation function to combine the features of neighboring nodes, GATs allow for a more flexible and adaptive approach by learning the importance of each neighbor during the aggregation process. The core of GATs is the attention network.

Attention networks

An **attention network**, often referred to as an **attention mechanism** or **attention model**, is a powerful concept in machine learning and artificial intelligence, particularly in the field of neural networks. It's inspired by how human attention works – focusing on specific parts of input data while processing information.

An attention mechanism allows the model to dynamically focus on different parts of the input data, assigning varying degrees of importance or attention to each part. This enables the model to weigh the relevance of different inputs when making predictions or decisions.

Attention networks are commonly used in tasks involving sequential data, such as **natural language processing** (**NLP**) tasks such as machine translation, text summarization, and sentiment analysis. In these tasks, the model needs to process sequences of words or tokens and understand the contextual relationships between them. By using attention mechanisms, the model can effectively capture long-range dependencies and attend to relevant parts of the input sequence.

Here's how an attention mechanism is leveraged in graph learning:

- In GATs, the attention mechanism is used to compute attention coefficients between a node and its neighbors.
- Attention coefficients represent the importance of each neighbor's features to the target node.
- The attention mechanism is typically implemented using an MLP or a single-layer neural network.
- Attention coefficients are computed based on the learned weights of the attention mechanism and the features of the target node and its neighbors.

Let's look at how we can compute the attention coefficient in a graph setting.

Attention coefficients computation

For each node, the attention coefficients are computed for all its neighbors. The attention coefficient between node *i* and its neighbor *j* is calculated as follows:

$$\alpha_{ij} = \frac{\exp\left(LeakyReLU\left(W^T \cdot \left[h_i \parallel h_j\right]\right)\right)}{\left(\sum_{k=1}^{n} \exp\left(LeakyReLU\left(W^T \cdot \left[h_i | h_k\right]\right)\right)\right)}$$

- || is the concatenation operation, and *LeakyReLU* is the leaky ReLU activation function.
- h_i and h_j are the learned node embeddings for nodes *i* and *j*, respectively.
- *W* is a learnable attention weight vector that is shared across all nodes.

The attention coefficients are normalized using the `softmax` function to ensure they sum up to 1 for each node.

Now that we understand how to compute attention coefficients, let's see how these are utilized in the aggregation step of a GNN.

Aggregation of neighbor features

Once the attention coefficients are computed, the features of the neighboring nodes are aggregated using a weighted sum.

The aggregated features for node *i* are calculated as follows:

$$h_i' = \sigma\left(\sum_{j \in N_i} \alpha_{ij} * h_j\right)$$

Here, σ is a non-linear activation function, such as ReLU. The aggregated features h_i' represent the updated representation of node *i* after considering the importance of its neighbors.

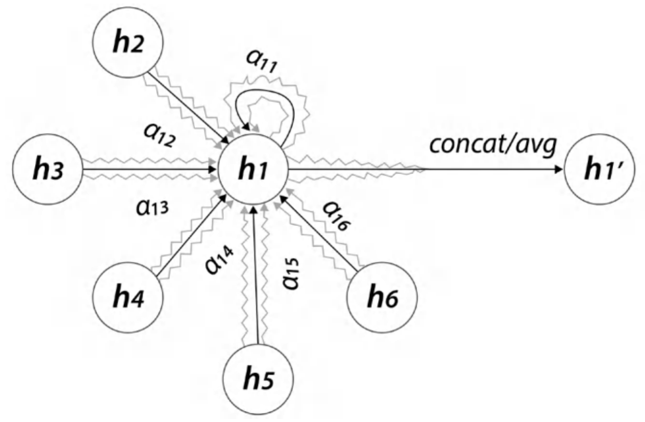


Figure 4.3 – Multi-head attention (with K = 3 heads) by node 1 on its neighborhood

To capture multiple aspects of the relationships between two nodes, we can leverage the concept of multi-head attention.

Multi-head attention

In GNNs, **multi-head attention** can be applied to learn representations of nodes as an extension of vanilla attention. Each "head" can be seen as a different perspective or attention mechanism applied to a node's neighborhood. By running multiple heads in parallel, the GNN can capture diverse aspects of the node's local graph structure and feature space. This allows the model to aggregate information from neighboring nodes in multiple ways, enhancing its ability to learn informative node representations that incorporate various patterns and relationships within the graph.

There are several notable aspects to keep in mind for multi-head attention:

- GATs can employ multi-head attention to capture different aspects of the node relationships.
- In multi-head attention, multiple attention mechanisms are used in parallel, each with its own set of learnable parameters.

- The output features from each attention head are concatenated or averaged to obtain the final node representations.
- Multi-head attention allows the model to learn diverse patterns and capture different types of dependencies between nodes.

At times, a single attention layer might be unable to capture complex relationships in a graph. Stacking multiple layers can help improve the learning space.

Stacking GAT layers

We can also use multiple GAT layers, similar to GCNs:

- Like GCNs, GAT layers can be stacked to capture higher-order dependencies and learn more abstract representations of the graph.
- In each layer, the updated node representations from the previous layer serve as input to the next layer.
- The final node representations obtained after multiple GAT layers can be used for downstream tasks such as node classification or graph classification.

GATs seamlessly combine the advantages of an **adaptive receptive field** and an **interpretable attention mechanism**, making them powerful tools for processing graph-structured data. The adaptive receptive field of GATs allows nodes to dynamically adjust their focus on relevant neighbors during information aggregation. Importantly, GATs provide interpretable attention coefficients, enabling a clear understanding of the model's decision-making process. The transparency in attention weights allows for intuitive insights into which neighbors contribute significantly to a node's representation, fostering interpretability and facilitating model debugging.

This combination of adaptability and interpretability makes GATs effective in capturing fine-grained local information while maintaining a global perspective, contributing to their success in various graph-based tasks.

Let's look at the model code for GAT:

```
class GAT(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels,
                 out_channels, heads=8, dropout=0.6):
        super(GAT, self).__init__()
        self.conv1 = GATConv(
            in_channels, hidden_channels,
            heads=heads, dropout=dropout)
        self.conv2 = GATConv(
            hidden_channels * heads, out_channels,
            heads=1, dropout=dropout)
    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.elu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

In this code, the GAT model class is defined using two `GATConv` layers. The first layer has multiple attention heads specified by the `heads` parameter, while the second layer has a single attention head. The activation function used is the **exponential linear unit** (**ELU**) function.

Note that in the `__init__` method of the GAT class, we multiply `hidden_channels` by `heads` when specifying the input channels for the second `GATConv` layer. This is because the output of the first layer has `hidden_channels * heads dimensions` due to the multiple attention heads.

Summary

In this chapter, we provided a comprehensive overview of graph-based deep learning models, starting with the fundamental concept of message passing and then delving into specific GNN architectures such as GCNs, GraphSAGE, and GATs.

Graph-based models rely on message passing, a key operation where nodes exchange information with neighbors to update their representations. GCNs perform convolutions on graphs, aggregating neighboring node information to learn node representations. GraphSAGE efficiently generates embeddings for large-scale graphs through neighborhood sampling. GATs integrate attention mechanisms, enabling nodes to assign varying importance weights to neighbors during message passing. These techniques enhance the capacity of graph-based models to capture complex relationships and patterns within data structures.

Building upon the foundational understanding of prevalent graph learning algorithms, we'll explore the contemporary challenges confronting GNNs in the upcoming chapter.

5
Graph Deep Learning Challenges

As deep learning on graphs has gained significant attention in recent years, researchers and practitioners have encountered numerous challenges that complicate the application of traditional deep learning techniques to graph data.

This chapter aims to give you a comprehensive overview of key challenges faced in graph learning, spanning from fundamental data issues to advanced model architectures and domain-specific problems. We will explore how the unique properties of graphs—such as their irregular structure, variable size, and complex dependencies—pose significant hurdles for conventional machine learning approaches.

By addressing these challenges, we aim to provide you with a solid foundation for understanding the current limitations and future directions of deep learning with respect to graphs. This chapter will serve as a roadmap, highlighting areas that require further investigation and innovation to advance the field of graph learning.

As we delve into each of these challenges, we will discuss current approaches, limitations, and potential avenues for future research. Understanding these challenges is crucial for developing more robust, efficient, and effective graph learning algorithms and applications.

The challenges discussed in this chapter can be broadly categorized into several key areas:

- Data-related challenges
- Model architecture challenges
- Computational challenges
- Task-specific challenges
- Interpretability and explainability

Data-related challenges

Graph data presents unique challenges due to its inherent complexity and diverse nature. In this section, we explore three key data-related challenges that significantly impact the development and application of graph learning algorithms.

Heterogeneity in graph structures

Graphs in different domains can have vastly different structural properties:

- **Node and edge types**: Many real-world graphs are heterogeneous, containing multiple types of nodes and edges. For instance, in an academic network, *nodes* could represent authors, papers, and conferences, while *edges* could represent authorship, citations, or attendance.
- **Attribute diversity**: Nodes and edges may have associated attributes of various types (numerical, categorical, textual), adding another layer of complexity to the learning process.
- **Structural variations**: Graphs can exhibit different global structures (for example, scale-free, small-world, random) and local patterns (for example, communities, motifs), requiring models that can adapt to these variations.

Dynamic and evolving graphs

Many real-world graphs are not static but change over time:

- **Temporal evolution**: Nodes and edges may appear or disappear over time, changing the graph structure dynamically. In a social media platform, the network of user connections evolves constantly as new friendships form and others dissolve. For instance, a user might connect with new colleagues after starting a job while losing touch with old classmates, causing nodes and edges to appear and disappear over time.
- **Attribute changes**: Node and edge attributes may also change over time, reflecting evolving properties or states. On a professional networking site such as LinkedIn, user profiles and connections undergo frequent updates. A user might change their job title, add new skills, or relocate, altering node attributes. Similarly, the strength of connections between professionals might increase as they collaborate on more projects, modifying edge attributes dynamically.
- **Concept drift**: Underlying patterns or rules governing the graph structure may change, requiring models that can adapt to these shifts. In an e-commerce recommendation system, the underlying patterns of user preferences can shift over time. Initially, the system might suggest products based on similar categories, but as consumer behavior evolves toward prioritizing sustainability or ethical sourcing, the recommendation algorithm needs to adapt its rules to reflect these changing preferences.

- **Streaming data**: In some applications, graph data arrives as a continuous stream, necessitating online learning algorithms that can process and update models incrementally. A real-time fraud detection system for a bank processes transaction data as a continuous stream. Each new transaction creates a node in the graph, instantly connecting to account holders and merchants. The system must analyze this incoming data on the fly, updating the graph structure and running fraud detection algorithms without interruption, all while adapting to emerging patterns of fraudulent behavior.

Noisy and incomplete graph data

Real-world graph data often suffers from quality issues:

- **Missing data**: Graphs may have missing nodes, edges, or attributes due to data collection limitations or privacy concerns.
- **Noisy connections**: Some edges in the graph may be erroneous or irrelevant, potentially misleading learning algorithms.
- **Uncertain attributes**: Node and edge attributes may be uncertain, imprecise, or subject to measurement errors.
- **Sampling bias**: The observed graph may be a biased sample of a larger population, leading to potential inaccuracies in learned models.

Addressing these data-related challenges is crucial for developing robust and effective graph learning algorithms. Practitioners must consider these issues when designing models, choosing evaluation metrics, and interpreting results. Future advancements in graph learning will likely focus on developing techniques that can handle larger, more complex, and dynamic graphs while being resilient to noise and incompleteness in the data.

Let's look into major architecture challenges faced by modern **graph neural networks** (**GNNs**).

Model architecture challenges

GNNs have shown remarkable success in various graph learning tasks. However, they face several architectural challenges that limit their effectiveness in certain scenarios. Here, we investigate four key model architecture challenges in graph learning.

Capturing long-range dependencies

GNNs often struggle to capture dependencies between distant nodes in the graph, as information typically propagates only to immediate neighbors in each layer. For instance, in scenarios such as citation networks, a paper might be influenced by another paper several citation links away. Standard GNNs might fail to capture this influence if it extends beyond their receptive field.

Graph attention mechanisms and **higher-order graph convolutions** represent two sophisticated approaches to enhancing GNNs' long-range capabilities. Graph attention mechanisms introduce a dynamic weighting system that allows the model to intelligently focus on the most significant connections within the graph, particularly those spanning longer distances. By assigning learnable weights to neighboring nodes, these mechanisms enable the model to automatically identify and prioritize influential nodes, even when they are distant in the graph structure.

This is complemented by higher-order graph convolutions, which take the traditional concept of graph convolution a step further. Instead of being limited to immediate neighbors, these advanced convolutions can process and aggregate information from nodes that are multiple hops away in a single operation. This means the model can directly capture complex relationships and patterns that exist across extended neighborhoods, leading to a more comprehensive understanding of the graph's structure and underlying relationships. Together, these approaches significantly improve the model's ability to process and understand complex graph-structured data by effectively managing both local and global information flow.

Depth limitation in GNNs

Unlike traditional **deep neural networks** (**DNNs**), GNNs often do not benefit from increased depth and may even suffer performance degradation with too many layers. This issue is noticeable in tasks such as molecule property prediction, where a deep GNN might not perform better than a shallow one, limiting the model's ability to learn complex hierarchical features of the molecular structure.

To overcome this limitation, several architectural modifications have been developed:

- **Residual connections**, inspired by **Residual Network** (**ResNet**) architectures in **computer vision** (**CV**), allow information to skip intermediate layers, facilitating gradient flow in deeper networks. These connections can be implemented by adding the input of each layer to its output, enabling the network to learn residual functions.
- **Jump connections** extend this concept by allowing information to jump across multiple layers, providing shorter paths for gradient propagation. This can be achieved through techniques such as **Jumping Knowledge Networks** (**JKNets**).
- **Adaptive depth mechanisms** dynamically adjust the effective depth of the network for each node, allowing different parts of the graph to be processed at different depths as needed. This approach can be implemented using techniques such as **DropEdge**, which stochastically removes edges or layers during training to create networks of varying effective depths.

Over-smoothing and over-squashing

As the number of GNN layers increases, node representations tend to converge to similar values (**over-smoothing**), and information from distant nodes may be "*squashed*" as it propagates through the graph (**over-squashing**). In a protein-protein interaction network, for instance, over-smoothing might cause the model to lose distinctive features of individual proteins, while over-squashing could prevent information about important distant interactions from influencing the final representation.

To combat these issues, several techniques have been proposed:

- **Normalization** methods such as **PairNorm** (`https://arxiv.org/abs/1909.12223`) or **DiffGroupNorm** (`https://arxiv.org/abs/2006.06972`) help maintain the diversity of node representations across layers by normalizing pairwise distances between node features. These methods adjust the scale of node representations to prevent them from converging to a single point.
- **Adaptive edge pruning** techniques dynamically remove less important edges during message passing, reducing redundant information flow and mitigating over-smoothing. This can be implemented using attention mechanisms or learned edge importance scores.
- **Hierarchical pooling** strategies progressively coarsen the graph, reducing its size while preserving its global structure. Methods such as **DiffPool** can be used to create hierarchical representations that capture information at different scales, helping to prevent over-squashing by providing more direct paths for information flow between distant nodes.

Balancing local and global information

GNNs need to effectively combine local structural information with global graph properties, but finding the right balance is often difficult. This challenge is evident in tasks such as traffic prediction on a road network, where the model needs to consider both the immediate surroundings of a road segment (*local*) and overall traffic flow patterns in the city (*global*).

To achieve this balance, several approaches have been developed:

- **Graph pooling techniques** aggregate node information hierarchically, creating a multi-scale representation of the graph. Methods such as **Self-Attention Graph Pooling** (**SAGPool**) or **TopKPool** use learnable pooling operations to select and combine important nodes at each level of the hierarchy.
- **Combining local GNN layers with global readout functions** allows the model to explicitly incorporate graph-level information. This can be achieved by using techniques such as **Set2Set** or **SortPool** to create fixed-size graph representations that capture global structure. Set2Set is a recurrent-based method that aggregates node representations by iteratively applying attention mechanisms, ensuring a dynamic and order-invariant set representation. SortPool, on the other hand, sorts node embeddings based on a chosen criterion (for example, node degree) and

then selects the top-*k* nodes to form a fixed-size graph representation. Both methods help in summarizing entire graphs while maintaining important structural information, thus ensuring better performance in tasks requiring both local and global understanding of the graph.

- **Attention mechanisms** that span different scales, such as those used in graph transformer architectures, allow the model to selectively focus on both local and global graph properties. These mechanisms can be implemented using **multi-head attention** (**MHA**), where different heads can attend to information at different scales, from immediate neighbors to distant nodes or even global graph properties.

Facing a model architecture challenge – an example

To illustrate these challenges with a concrete example, let's consider a large social network analysis task.

Imagine developing a GNN model to predict user interests on a platform such as X (formerly Twitter). The graph consists of millions of users (*nodes*) connected by follower relationships (*edges*), with tweets and hashtags as additional features. The following are some challenges that may arise:

- **Long-range dependencies**: The model needs to capture influences from popular users or trending topics that might be several hops away in the follower graph.
- **Depth limitation**: Simply stacking more GNN layers doesn't necessarily improve the model's ability to understand complex user interaction patterns.
- **Over-smoothing and over-squashing**: With a deep GNN, users with diverse interests might end up with similar representations, losing important distinctions. Information about niche interests from distant parts of the network might get lost as it propagates through the graph.
- **Balancing local and global information**: The model must combine a user's immediate network (*local*) with platform-wide trends and influential users (*global*) to make accurate interest predictions.

By addressing these architectural challenges using the strategies mentioned previously, practitioners can develop more powerful and flexible GNN models capable of handling a wide range of graph learning tasks across various domains.

As GNNs continue to evolve and find applications in increasingly large-scale graph datasets, practitioners face significant computational challenges that push the boundaries of existing algorithms and computing infrastructures. The following section explores three primary computational challenges that researchers and developers must navigate to unlock the full potential of GNNs across various domains.

Computational challenges

As graph learning techniques continue to evolve and find applications in increasingly complex domains, they face significant computational hurdles. The sheer scale and intricacy of real-world graphs pose formidable challenges to existing algorithms and computing infrastructures. Here, we delve into three primary computational challenges that researchers and practitioners encounter when working with large-scale graph data: scalability issues, memory constraints, and the need for parallel and distributed computing solutions.

Scalability issues for large graphs

As graph data continues to grow in size and complexity, scalability has become a critical challenge for graph learning algorithms. This issue is particularly evident in scenarios such as social network analysis or web-scale graphs, where billions of nodes and edges are common. Traditional graph algorithms often have time complexities that scale poorly with graph size, making them impractical for large-scale applications.

To address this challenge, several approaches have been developed.

Sampling-based methods, such as **GraphSAGE**, which we explored in *Chapter 4*, or **FastGCN**, reduce computational complexity by operating on subsets of the graph. These techniques randomly sample neighborhoods or nodes during training, allowing the model to scale to large graphs by approximating full-graph computations.

Another approach is the use of simplified propagation rules, as seen in models such as **Simple Graph Convolution** (**SGC**) or **Scalable Inception Graph Neural Networks** (**SIGNs**). These methods reduce the number of nonlinear operations and parameter updates required in each iteration, significantly speeding up training and inference on large graphs.

Memory constraints in graph processing

Processing large graphs often requires holding substantial amounts of data in memory, which can exceed the capacity of single machines. This challenge is particularly acute in tasks involving large knowledge graphs or molecular datasets with millions of compounds. To address this, several techniques have been developed:

- **Out-of-core computing techniques**, such as those used in **GraphChi** or **X-Stream**, allow the processing of graphs that don't fit in main memory by efficiently managing data on disk. These methods carefully organize graph data and computations to minimize random access to secondary storage.
- **Graph compression techniques**, such as those employed in **k2-tree** representations, reduce the memory footprint of large graphs by exploiting structural regularities and redundancies. These approaches can significantly reduce storage requirements while still allowing efficient query operations.

- Another effective approach is the use of **mini-batch training strategies**, as seen in **Cluster-GCN** (`https://arxiv.org/abs/1905.07953`) or **GraphSAINT** (`https://arxiv.org/abs/1907.04931`). These methods process the graph in small, manageable subgraphs or batches, allowing training on much larger graphs than would be possible with full-graph approaches.

Parallel and distributed computing for graphs

The scale of many real-world graphs necessitates the use of parallel and distributed computing techniques to achieve reasonable processing times. This is crucial in applications such as analyzing internet-scale networks or processing large-scale scientific simulation data:

- **Graph-parallel frameworks**, such as **Pregel**, **GraphLab**, or **PowerGraph**, provide programming models specifically designed for distributed graph computations. These frameworks often use a **"think like a vertex" paradigm**, where computations are expressed from the perspective of individual nodes, facilitating parallelization across a cluster of machines.
- **Distributed GNN training techniques**, such as those used in **PinSage** or **AliGraph**, allow GNN models to be trained on massive graphs spread across multiple machines. These approaches often combine **data parallelism** (distributing the graph across machines) with **model parallelism** (distributing the **neural network** (**NN**) itself).
- **GPU-accelerated graph processing**, exemplified by frameworks such as **Gunrock** or **cuGraph**, leverages the massive parallelism of GPUs to speed up graph algorithms. These approaches often require careful algorithm redesign to match GPU architectures, such as using warp-centric programming models or optimizing memory access patterns.

By addressing these computational challenges, we can develop graph learning systems capable of handling the scale and complexity of real-world graph data, opening up new possibilities for applications in areas such as social network analysis, recommender systems, and scientific computing.

While addressing these broad computational challenges is crucial, it's equally important to consider specific issues that arise in different graph-related tasks. Let's explore some of these task-specific challenges, starting with node classification in imbalanced graphs.

Task-specific challenges

While graph learning algorithms face general challenges, certain tasks present unique difficulties that require specialized approaches. In this section, we consider four common task-specific challenges in graph learning, each with its own set of complexities and proposed solutions.

Node classification in imbalanced graphs

Node classification in real-world graphs often suffers from class imbalance, where some classes are significantly underrepresented. This issue is prevalent in scenarios such as fraud detection in financial transaction networks, where fraudulent transactions are typically rare compared to legitimate ones. The imbalance can lead to biased models that perform poorly on minority classes.

Some approaches to mitigate this include the following:

- **Re-sampling techniques**, such as over-sampling minority classes or under-sampling majority classes, can be adapted for graph data. For instance, **GraphSMOTE** extends the **Synthetic Minority Over-sampling TEchnique** (**SMOTE**) algorithm to graph-structured data, generating synthetic samples for minority classes by interpolating between existing nodes in the feature and graph space.
- **Cost-sensitive learning approaches** assign higher penalties for misclassifications of minority class nodes during training. This can be implemented by modifying the loss function to weight errors on minority classes more heavily.
- Another effective approach is to use **meta-learning techniques**, such as **few-shot learning** (**FSL**) algorithms adapted for graphs. These methods, such as **Meta-GNN** or **graph prototypical networks** (**GPNs**), aim to learn generalizable representations that can perform well on underrepresented classes with limited samples.

Link prediction in sparse graphs

Link prediction in sparse graphs presents unique challenges, as the vast majority of potential edges are absent, leading to extreme class imbalance in the link prediction task. This is common in biological networks, where only a small fraction of possible interactions between entities are observed. The sparsity can make it difficult to learn meaningful patterns.

To tackle this issue, several specialized approaches have been proposed. **Negative sampling** techniques carefully select a subset of non-existent edges as negative examples during training, balancing the dataset without introducing too much noise. Advanced methods such as **knowledge-based generative adversarial networks** (**KBGANs**) use adversarial training to generate high-quality negative samples.

Graph generation and reconstruction

Graph generation and reconstruction tasks aim to create new graphs or complete partial graphs, which is challenging due to the discrete nature of graphs and the need to maintain complex structural properties. This is crucial in applications such as drug discovery, where generating valid molecular graphs is essential.

One major approach to this challenge is the use of **variational autoencoders** (**VAEs**) adapted for graphs, such as **GraphVAE** or **variational graph autoencoders** (**VGAE**). These models learn a continuous latent space representation of graphs, allowing for the generation of new graphs by sampling from this space.

However, ensuring the validity of generated graphs remains a challenge. Another powerful approach is the use of autoregressive models for graph generation, as seen in **GraphRNN** or **graph recurrent attention networks** (**GRANs**). These models generate graphs sequentially, one node or edge at a time, capturing complex dependencies in the graph structure.

Graph matching and alignment

Graph matching and alignment involve finding correspondences between nodes of different graphs, which is crucial in tasks such as network alignment in systems biology or matching 3D objects in CV. This task is computationally challenging due to the combinatorial nature of the problem and the need to consider both structural and attribute similarities.

To overcome this challenge for larger graphs, approximate methods based on graph embeddings have gained popularity. Models such as **REpresentation learning-based Graph Alignment** (**REGAL**) learn node embeddings that preserve local and global graph structure, allowing for efficient alignment by matching nodes in the embedding space. Recent advances also include the application of GNNs to the matching task, such as the **cross-graph attention network**, which learns to match nodes by attending to their local neighborhoods across graphs. By addressing these task-specific challenges, we can develop more effective and robust graph learning models tailored to the unique requirements of different applications. As the field progresses, we can expect to see further innovations that combine insights from multiple approaches to tackle these complex problems.

As we move from discussing technical innovations in graph learning, it's crucial to consider how these complex models can be understood and explained, especially when applied to high-stakes domains. This leads us to our next important topic: the interpretability and explainability of graph learning models.

Interpretability and explainability

As graph learning models become increasingly complex and are applied to critical domains such as healthcare, finance, and social sciences, the need for interpretable and explainable models has grown significantly. Here, we explore two key aspects of interpretability and explainability in graph learning.

Explaining GNN decisions

GNNs often act as black boxes, making it challenging to understand why they make certain predictions. This lack of transparency can be problematic in high-stakes applications such as drug discovery or financial fraud detection. To address this, several approaches have been developed to explain GNN decisions:

- One prominent method is **GNNExplainer**, which identifies important subgraphs and features that influence a model's predictions. It does this by optimizing a mutual information objective between a conditional distribution of the GNN's predictions and a simplified explanation.
- Another approach is **GraphLIME**, an extension of the **Local Interpretable Model-agnostic Explanation** (**LIME**) framework for graph-structured data. It explains individual predictions by learning an interpretable model locally around the prediction.
- Gradient-based methods, such as **Grad-CAM** adapted for graphs, provide explanations by visualizing the gradients of the output with respect to intermediate feature maps, highlighting important regions of the input graph. Some recent works also focus on counterfactual explanations for GNNs, generating minimal changes to the input graph that would alter the model's prediction.

These approaches help in understanding model decisions and identify potential biases or vulnerabilities in the model.

Visualizing graph embeddings

Graph embeddings, which represent nodes or entire graphs as vectors in a low-dimensional space, are fundamental to many graph learning tasks. However, interpreting these embeddings can be challenging due to their high-dimensional nature. Various techniques have been developed to visualize and understand these embeddings:

- **Dimensionality reduction techniques**, such as **t-distributed Stochastic Neighbor Embedding** (**t-SNE**) or **Uniform Manifold Approximation and Projection** (**UMAP**), are commonly used to project high-dimensional embeddings into 2D or 3D spaces for visualization. These methods aim to preserve local relationships between points, allowing for the identification of clusters and patterns in the embedding space.
- **Interactive visualization tools**, such as **TensorBoard Projector** or **Embedding Projector**, allow users to explore embeddings dynamically, zooming in on specific regions and examining relationships between nodes. Some advanced approaches combine embedding visualization with the original graph structure. For instance, **GraphTSNE** integrates graph structural information into the t-SNE algorithm, producing layouts that reflect both the embedding similarity and the graph topology.

- Another innovative approach is the use of **graph generation techniques** to visualize embeddings. By training a graph generative model on embeddings and original graphs, one can generate synthetic graphs that represent different regions of the embedding space, providing intuitive visualizations of what the embeddings have captured.

By addressing these aspects of interpretability and explainability, researchers aim to bridge the gap between the performance of complex graph learning models and the need for transparent, trustworthy AI systems. As the field progresses, we can expect to see further integration of these techniques into mainstream graph learning frameworks, making them more accessible to practitioners across various domains. The development of interpretable and explainable graph learning models not only enhances trust and adoption but also opens new avenues for scientific discovery and knowledge extraction from complex graph-structured data.

Summary

In this chapter, we have explored the multifaceted challenges that define the current landscape of graph learning. From fundamental issues of handling large-scale, heterogeneous, and dynamic graph data to intricate problems of designing effective GNN architectures, each challenge presents unique obstacles and opportunities for innovation. We've examined the computational hurdles of processing massive graphs, nuanced difficulties in specific tasks such as node classification and link prediction, and the growing demand for interpretable and explainable models.

These challenges are not isolated; they intersect and compound each other, creating a complex ecosystem of problems that researchers and practitioners must navigate. As graph learning continues to evolve and find applications in critical domains such as healthcare, finance, and social sciences, addressing these challenges becomes not just an academic pursuit but a practical necessity.

The future of graph learning lies in developing holistic solutions that can handle the scale, complexity, and dynamism of real-world graphs while providing robust, efficient, and interpretable models. By confronting these challenges head-on, you are now poised to unlock new possibilities and drive innovations that can transform how we understand and interact with the interconnected world around us.

As we look to the future of graph learning, one promising avenue is the integration of **large language models** (**LLMs**) with graph-based approaches. The next chapter explores how LLMs can be leveraged to enhance graph learning techniques, potentially addressing some of the challenges discussed here while opening up new possibilities for more sophisticated graph analysis and understanding.

6
Harnessing Large Language Models for Graph Learning

Traditionally, **graph neural networks** (**GNNs**) have been the workhorse for graph learning tasks, achieving impressive results. However, recent research explores the exciting potential of **large language models** (**LLMs**) in this domain.

In this chapter, we'll delve into the intersection of LLMs and graph learning, exploring how these powerful language models can enhance graph-based tasks. We'll begin with an overview of LLMs, followed by a discussion of the limitations of GNNs and the motivations for incorporating LLMs. Then, we'll explore various approaches for utilizing LLMs in graph learning, the intersection of **retrieval-augmented generation** (**RAG**) with graphs, and explain the advantages and challenges associated with this integration.

In this chapter, we'll explore the following topics:

- Understanding LLMs
- Textual data in graphs
- LLMs for graph learning
- Integrating RAG with graph learning
- Challenges in integrating LLMs

Understanding LLMs

LLMs are a significant advancement in **artificial intelligence** (**AI**), particularly in **natural language processing** (**NLP**) and understanding. These models are designed to understand, generate, and interact with human language in a way that's both meaningful and contextually relevant. The development and evolution of LLMs have been marked by a series of innovations that have expanded their capabilities and applications across various domains.

At their core, LLMs are trained on vast datasets of text from the internet, books, articles, and other sources of written language. This training involves analyzing patterns, structures, and the semantics of language, enabling these models to generate coherent, contextually appropriate text based on the input they receive. The training process relies on deep learning techniques, particularly neural networks, which allow the models to improve their language capabilities over time through exposure to more data.

One of the key characteristics of LLMs is their size, which is measured in the number of parameters they contain. Early models had millions of parameters, but the most advanced models today boast tens or even hundreds of billions of parameters. This increase in size has been correlated with a significant improvement in the models' ability to understand and generate human-like text, making them more effective for a wide range of applications.

LLMs have a wide array of applications, from simple tasks such as grammar correction and text completion to more complex ones such as writing articles, generating code, translating languages, and even creating poetry or prose. They're also used in conversational agents, providing the backbone for chatbots and virtual assistants, which can engage in more natural and meaningful interactions with users.

The evolution of LLMs has been marked by significant milestones:

- **1990s**: The era of **statistical language models** (**SLMs**) began, utilizing **n-gram models** to predict the next word in a sequence based on a few preceding words. These models faced challenges with high-order predictions due to data sparsity issues.
- **Early 2000s**: The introduction of **neural language models** (**NLMs**), which employed neural networks such as **multilayer perceptrons** (**MLPs**) and **recurrent neural networks** (**RNNs**), marked a shift toward understanding deeper linguistic relationships.
- **2010s**: The development of word embeddings such as **Word2Vec** and **GloVe**, which represent words in continuous vector spaces, allowed models to capture semantic meaning and context.
- **2017**: Transformer architecture was introduced, leading to a breakthrough in handling sequential data without the need for recurrent processing. This architecture is the foundation of many subsequent LLMs.
- **2018**: The **Generative Pre-Trained Transformer** (**GPT**) was released by OpenAI, showcasing the power of transformers in language understanding and generation.
- **2019**: **Bidirectional Encoder Representations from Transformers** (**BERT**) by Google revolutionized the field by introducing a model trained to understand the context from both directions in a piece of text.
- **2020s**: Even larger models began to emerge, such as **GPT-3**, which demonstrated remarkable capabilities in generating human-like text, and models that can integrate LLMs with other AI fields were introduced, such as graph learning.

- **2023 – 2024**: The landscape saw an explosion of powerful LLMs, starting with ChatGPT, followed by **GPT-4**, Claude from Anthropic, Gemini from Google, and Llama from Meta, indicating a trend toward more specialized and powerful language models.

The remarkable achievements of LLMs have sparked interest in harnessing their capabilities for tasks in graph machine learning. On the one hand, the extensive knowledge and logical prowess of LLMs offer promising prospects to improve upon conventional GNN models. On the other hand, the organized representations and concrete knowledge embedded within graphs hold the potential to mitigate some of the primary shortcomings of LLMs, including their tendency to generate misleading information and their challenges with interpretability.

Textual data in graphs

One of the fundamental hurdles in deploying GNNs lies in acquiring sophisticated feature representations for nodes and edges. This becomes particularly crucial when these elements are associated with complex textual attributes such as descriptions, titles, or abstracts.

Traditional methods, such as the bag-of-words approach or the utilization of pre-trained word embedding models, have been the norm. However, these techniques typically fall short of grasping the subtle semantic intricacies inherent in the text. They tend to overlook the context and the interdependencies between words, leading to a loss of critical information that could be essential for the GNN to perform optimally.

To overcome this challenge, there's a growing need for more advanced methods that can understand and encode the richness of language into the graph structure. This is where LLMs come into play. With their deep understanding of language nuances and context, LLMs can generate embeddings that capture a broader spectrum of linguistic features.

By integrating LLMs into the feature extraction process for GNNs, you can potentially encode richer, more informative representations that reflect the true semantic content of the textual attributes, thereby enhancing the GNN's ability to perform tasks such as node classification, link prediction, and graph generation with greater accuracy and efficiency.

LLMs have capabilities that extend beyond generating textual embeddings as features. LLMs are good at generating augmented information from original text attributes. They can be used to generate tags/labels and other useful metadata in an unsupervised/semi-supervised way.

A significant benefit of LLMs is their capacity to adapt to new tasks with minimal or no labeled data, owing to their extensive pre-training on broad text datasets. This ability for few-shot learning can help reduce the dependency of GNNs on extensive labeled datasets.

One strategy involves employing LLMs to directly predict outcomes for graph-related tasks by framing the graph's structure and the information of its nodes within natural language prompts. Techniques such as InstructGLM refine LLMs such as Llama and GPT-4 with well-crafted prompts that detail the graph's topology, including aspects such as node connections and neighborhoods. These optimized LLMs are capable of making predictions for tasks such as **node classification** and **link prediction** without requiring any labeled examples at the inference stage.

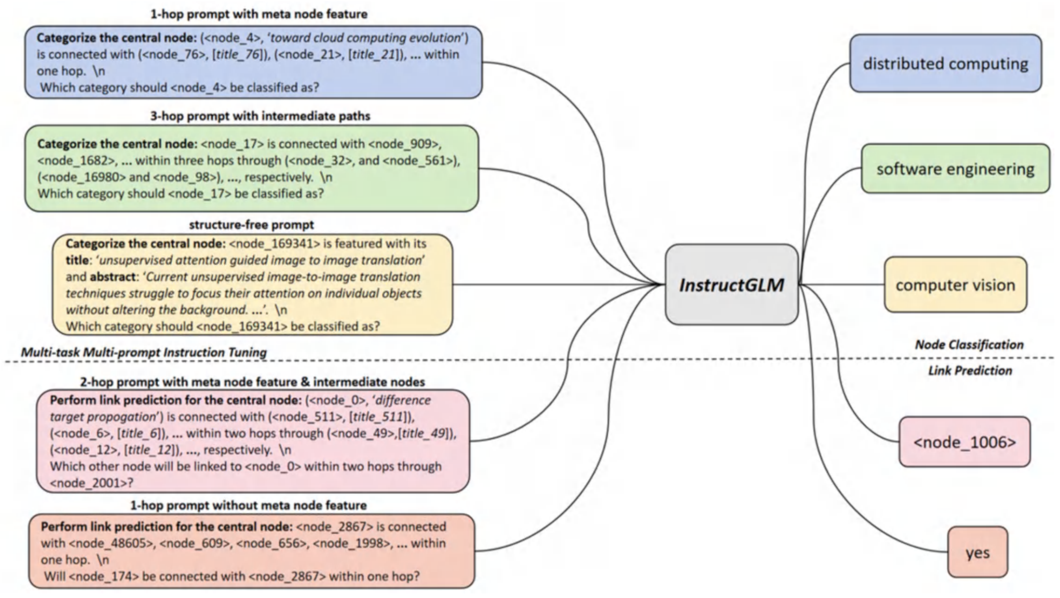
Leveraging InstructGLM

InstructGLM is a framework that leverages natural language to describe both graph structure and node features to a generative LLM, addressing graph-related problems through instruction-tuning. It's a proposed instruction fine-tuned **graph language model** (**GLM**) that utilizes natural language instructions for graph machine learning, offering a powerful NLP interface for graph-related tasks.

The InstructGLM framework involves a multi-task, multi-prompt instructional tuning process to refine LLMs and integrate them with graphs effectively. This approach aims to reduce the reliance on labeled data by utilizing self-supervised learning on graphs and leveraging LLMs as text encoders to enhance performance and efficiency.

The InstructGLM technique employs linguistic cues that articulate the patterns of connections and the characteristics of nodes within a graph. These prompts serve as a teaching mechanism, guiding LLMs to comprehend the intricate architecture and inherent meaning of graphs.

As shown in *Figure 6.1*, the InstructGLM framework presents a sophisticated approach to multi-task, multi-prompt instructional tuning for LLMs:


1-hop prompt with meta node feature
3-hop prompt with intermediate paths
structure-free prompt
Multi-task Multi-prompt Instruction Tuning
2-hop prompt with meta node feature & intermediate nodes
1-hop prompt without meta node feature
InstructGLM
distributed computing
software engineering
computer vision
Node Classification
Link Prediction
<node_1006>
yes

Categorize the central node: (<node_4>, '*toward cloud computing evolution*') is connected with (<node_76>, [*title_76*]), (<node_21>, [*title_21*]), ... within one hop. \n
Which category should <node_4> be classified as?

Categorize the central node: <node_17> is connected with <node_909>, <node_1682>, ... within three hops through (<node_32>, and <node_561>), (<node_16980> and <node_98>), ..., respectively. \n
Which category should <node_17> be classified as?

Categorize the central node: <node_169341> is featured with its **title**: '*unsupervised attention guided image to image translation*' and **abstract**: '*Current unsupervised image-to-image translation techniques struggle to focus their attention on individual objects without altering the background. ...*'. \n
Which category should <node_169341> be classified as?

Perform link prediction for the central node: (<node_0>, '*difference target propogation*') is connected with (<node_511>, [*title_511*]), (<node_6>, [*title_6*]), ... within two hops through (<node_49>,[*title_49*]), (<node_12>, [*title_12*]), ..., respectively. \n
Which other node will be linked to <node_0> within two hops through <node_2001>?

Perform link prediction for the central node: <node_2867> is connected with <node_48605>, <node_609>, <node_656>, <node_1998>, ... within one hop. \n
Will <node_174> be connected with <node_2867> within one hop?

Figure 6.1 – InstructGLM multi-task usage. Source: Ye et al., 2024 (https://arxiv.org/abs/2308.07134)

This figure illustrates the core components of InstructGLM, showcasing different types of prompts and their applications:

- **1-hop prompt with meta node feature**: This prompt type categorizes central nodes based on their immediate connections, as shown in the blue box at the top left of the figure.
- **3-hop prompt with intermediate paths**: Depicted in the green box, this prompt explores connections up to three hops away, providing a broader context for node classification.
- **Structure-free prompt**: The yellow box demonstrates how InstructGLM can categorize nodes based on their inherent features without relying on structural information.
- **2-hop prompt with meta node feature & intermediate nodes**: Illustrated in the pink box, this prompt type is used for link prediction tasks, considering connections up to two hops away.
- **1-hop prompt without meta node feature**: Another link prediction prompt, as shown in the orange box, this focuses on immediate connections without additional node information.

Figure 6.1 also highlights the dual focus of InstructGLM on node classification and link prediction tasks, as indicated by the dotted line separating these two primary functions.

Although utilizing LLMs as opaque predictors has been effective, their accuracy diminishes for more intricate graph tasks where detailed modeling of the structure proves advantageous. Consequently, some methods combine LLMs with GNNs, where the GNN maps out the graph structure, and the LLM enriches this with a deeper semantic understanding of the nodes based on their textual descriptions.

Next, let's see how LLMs can help us with graph learning.

LLMs for graph learning

Researchers have delved into various strategies for incorporating LLMs into the graph learning process. Each method presents distinct benefits and potential uses. Let's look at some of the key functions that LLMs can fulfill.

LLMs as enhancers

Traditional GNNs rely on the quality of initial node features, often with limited textual descriptions. LLMs, with their vast knowledge and language comprehension abilities, can bridge this gap. By enhancing these features, LLMs empower GNNs to capture intricate relationships and dynamics within the graph, leading to superior performance on tasks such as node classification or link prediction.

There are two primary methods for harnessing LLMs as enhancers. The first is **feature-level enhancement**, which can be achieved in various ways using LLMs:

- **Synonyms and related concepts**: The LLM goes beyond the surface level of the text description by recognizing synonyms and semantically related concepts. This helps capture a broader range of information that might not be explicitly mentioned. For instance, if a product description mentions *waterproof* and *hiking boots*, the LLM can infer that the product is suitable for outdoor activities.
- **Implicit relationships**: LLMs can extract implicit relationships from the text. These relationships can be crucial for understanding the node's context within the graph. For example, in a social network, the LLM might infer a friendship between two nodes based on their frequent interactions, even if the word *friend* is never explicitly mentioned.
- **External knowledge integration**: LLMs can access and integrate external knowledge bases to further enrich the node representation. This could involve linking product information to user reviews or connecting protein descriptions (in a biological network) to known protein-protein interactions.

The other method is **text-level enhancement**, which can be employed to create richer, more informative textual descriptions for nodes.

This approach focuses on creating entirely new textual descriptions for the nodes, which is particularly beneficial when the original descriptions are limited or lack context. The LLM acts as a content generator, leveraging information about the node and its surrounding context within the graph to create a new, more informative description. This context might include the following aspects:

- Original text description
- Labels of neighboring nodes
- The structure of the graph (for example, the number of edges connected to the node)

The LLM utilizes this information to generate a rich textual description that captures the relationships with neighboring nodes, the overall network structure, and any relevant external knowledge. This newly created description becomes the node's enhanced feature, providing the GNN with a more comprehensive understanding of the node's role within the graph.

Benefits and challenges of LLM enhancement

LLM-based enhancement offers several advantages for graph learning:

- **Improved feature representation**: The enhanced node features capture a richer and more nuanced understanding of the node's context within the graph. This allows GNNs to learn more complex relationships and patterns, leading to improved performance on tasks such as node classification, link prediction, and community detection.

- **Handling limited data**: LLMs can address situations where node descriptions are sparse or lack detail. By inferring relationships and leveraging external knowledge, they create more informative representations, mitigating the challenges associated with limited data availability.
- **Identifying implicit connections**: LLMs can go beyond the surface-level information in node features and identify subtle connections based on their understanding of language. This can be crucial for tasks such as uncovering hidden communities within a social network or predicting the existence of links between nodes in a biological network.

However, while LLM enhancement offers a compelling path forward, there are a few challenges to navigate:

- **The cost of computation**: Training and running LLMs can be computationally expensive, especially for massive graphs. Careful optimization strategies are needed to ensure scalability.
- **Data bias inheritance**: LLMs aren't immune to biases present in their training data. It's crucial to ensure the LLM that's used for enhancement is trained on high-quality, unbiased data to prevent skewed results.
- **The explainability enigma**: Understanding how LLMs generate enhanced features can be challenging. This lack of transparency can make it difficult to interpret the results of GNNs that utilize these features. Researchers are actively working on developing methods to make the enhancement process that's implemented by LLMs more transparent.

Real-world applications

Here are multiple real-world examples of how LLMs can enhance the graph learning process.

- **Drug discovery and bioinformatics**: In drug discovery, predicting adverse **drug-drug interactions** (**DDIs**) is crucial for patient safety. Traditional methods often struggle due to the sheer volume of possible drug combinations and interactions. GNNs can model these relationships by representing drugs as *nodes* and known interactions as *edges*. When enhanced with LLMs, which process biomedical literature to extract information about drug mechanisms, side effects, and interactions, these models become significantly more powerful. The LLM-generated embeddings enrich the node and edge features in the GNN, leading to more accurate predictions of potential DDIs and ultimately safer medication management.
- **Social network analysis**: Detecting misinformation on social media is another area where LLMs can enhance GNNs. GNNs can model social networks with *nodes* representing users and *edges* representing interactions such as likes, shares, and comments. By processing the content of posts, an LLM can extract themes, sentiments, and potentially misleading information, which are then integrated into the graph. This enrichment enables the GNN to better identify clusters of misinformation and predict which users are most likely to spread false information, facilitating more effective interventions to maintain information integrity.

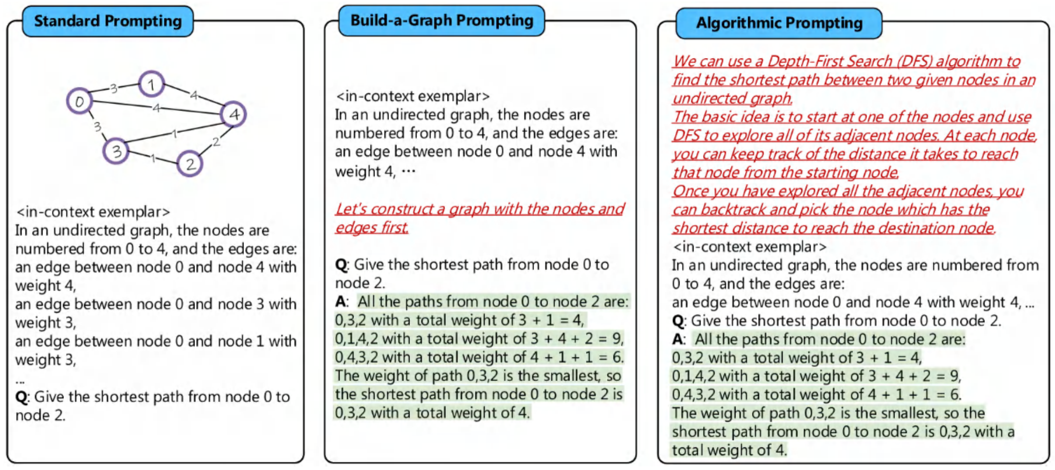
- **Financial fraud detection**: Financial fraud detection involves identifying suspicious patterns among millions of transactions. GNNs can represent these transactions as graphs, with *nodes* as accounts and *edges* as transactions. An LLM can analyze transaction descriptions and notes to extract keywords and patterns indicative of fraud, enhancing the GNN's node and edge features. This integration allows the GNN to detect fraudulent transactions more accurately by considering both transactional patterns and contextual information from textual descriptions, leading to more robust fraud detection systems.
- **Academic research and collaboration networks**: Identifying potential research collaborators is essential for advancing scientific discovery. GNNs can model academic networks, with *nodes* representing researchers and *edges* representing co-authorship or citation relationships. An LLM can analyze publication abstracts, keywords, and research interests, transforming these textual features into embeddings that are integrated into the graph. This enhancement enables the GNN to recommend potential collaborators by considering both structural relationships and semantic similarities in research interests, fostering more effective scientific collaborations.
- **Knowledge graph construction**: Building comprehensive knowledge graphs involves integrating information from diverse and often unstructured sources. GNNs can model knowledge graphs with *nodes* representing entities and *edges* representing relationships. An LLM can extract entities and relationships from textual data sources, such as news articles, scientific literature, and web pages, and use these insights to augment the knowledge graph with additional nodes and edges. This enhancement allows the GNN to build more complete and accurate knowledge graphs by incorporating detailed and contextually rich information from a wide array of textual sources, facilitating better knowledge representation and discovery.

The integration of LLMs with GNNs provides a powerful approach to enhancing various applications by incorporating rich, contextual information from textual data.

LLMs as predictors

With the advancement in language understanding, thanks to transformer-based LLMs, researchers are exploring how best to represent graph data in text for LLMs. A recent paper titled *Can Language Models Solve Graph Problems in Natural Language?* (`https://arxiv.org/html/2305.10037v3`) talks about an LLM constructing graphs from text descriptions, enhancing its graph comprehension. This paper demonstrates how prompting can help LLM understand graph structure and provide results using a set of instructions (algorithms) provided in the prompt.

Figure 6.2 shows three distinct approaches to graph analysis through prompting techniques:



Standard Prompting

<in-context exemplar>
In an undirected graph, the nodes are numbered from 0 to 4, and the edges are:
an edge between node 0 and node 4 with weight 4,
an edge between node 0 and node 3 with weight 3,
an edge between node 0 and node 1 with weight 3,
...
Q: Give the shortest path from node 0 to node 2.

Build-a-Graph Prompting

<in-context exemplar>
In an undirected graph, the nodes are numbered from 0 to 4, and the edges are:
an edge between node 0 and node 4 with weight 4, ···

Let's construct a graph with the nodes and edges first.

Q: Give the shortest path from node 0 to node 2.
A: All the paths from node 0 to node 2 are:
0,3,2 with a total weight of 3 + 1 = 4,
0,1,4,2 with a total weight of 3 + 4 + 2 = 9,
0,4,3,2 with a total weight of 4 + 1 + 1 = 6.
The weight of path 0,3,2 is the smallest, so the shortest path from node 0 to node 2 is 0,3,2 with a total weight of 4.

Algorithmic Prompting

We can use a Depth-First Search (DFS) algorithm to find the shortest path between two given nodes in an undirected graph.
The basic idea is to start at one of the nodes and use DFS to explore all of its adjacent nodes. At each node, you can keep track of the distance it takes to reach that node from the starting node.
Once you have explored all the adjacent nodes, you can backtrack and pick the node which has the shortest distance to reach the destination node.
<in-context exemplar>
In an undirected graph, the nodes are numbered from 0 to 4, and the edges are:
an edge between node 0 and node 4 with weight 4, ...
Q: Give the shortest path from node 0 to node 2.
A: All the paths from node 0 to node 2 are:
0,3,2 with a total weight of 3 + 1 = 4,
0,1,4,2 with a total weight of 3 + 4 + 2 = 9,
0,4,3,2 with a total weight of 4 + 1 + 1 = 6.
The weight of path 0,3,2 is the smallest, so the shortest path from node 0 to node 2 is 0,3,2 with a total weight of 4.

Figure 6.2 – Understanding graph structure using prompting. Source: Wang et al., 2024 (https://arxiv.org/html/2305.10037v3)

Note

The figure *Overview of Build-a-Graph prompting and Algorithmic prompting* (`https://arxiv.org/html/2305.10037v3`) by Wang et al. (2024) is licensed under CC BY 4.0.

This illustration demonstrates how LLMs can be guided to comprehend and solve graph-related problems using different prompting strategies:

- **Standard prompting**: The first column presents the standard prompting method. Here, a simple undirected graph is depicted with nodes numbered from 0 to 4. The prompt provides context by describing the graph's structure, including the weights of edges connecting various nodes. The question that's posed is to find the shortest path from node 0 to node 2, demonstrating a basic graph traversal problem.
- **Build-a-graph prompting**: The middle column introduces a more sophisticated approach called build-a-graph prompting. This method begins by instructing the LLM to construct the graph based on the given information. It then guides the model through the process of identifying all possible paths between nodes 0 and 2 and calculating their total weights. This step-by-step approach helps the LLM to systematically analyze the graph and determine the shortest path.

- **Algorithmic prompting**: The rightmost column showcases algorithmic prompting, which is the most advanced technique presented. It outlines a **depth-first search** (**DFS**) algorithm to find the shortest path between two nodes in an undirected graph. This method provides a detailed explanation of how to implement the algorithm, including tracking distances and backtracking to identify the optimal path. The prompt then applies this algorithm to the same graph problem, demonstrating how a more complex analytical approach can be used to solve graph traversal questions.

The **GPT4Graph** (`https://arxiv.org/pdf/2305.15066`) study used graph markup languages and self-prompting to improve LLM understanding before generating the final output. The main strategy involves inputting graph data, ensuring the LLM understands it, and then querying it. However, this method struggles with scalability for large graphs due to context length limitations and requires new prompts for new tasks. To address these challenges and improve graph learning capabilities, let's explore how to integrate RAG with graph learning.

Integrating RAG with graph learning

RAG is an AI framework that combines the power of LLMs with external knowledge retrieval to produce more accurate, relevant, and up-to-date responses. Here's how it works:

1. **Information retrieval**: When a query is received, RAG searches a knowledge base or database to find relevant information.
2. **Context augmentation**: The retrieved information is then used to augment the input to the language model, providing it with additional context.
3. **Generation**: The LLM uses this augmented input to generate a response that's both fluent and grounded in the information that's been retrieved.

RAG offers several compelling perks:

- **Improved accuracy**: By grounding responses in retrieved information, RAG reduces hallucinations and improves factual accuracy.
- **Up-to-date information**: The knowledge base can be updated regularly, allowing the system to access current information without having to retrain the entire model.
- **Transparency**: RAG can provide sources for its information, increasing trustworthiness and allowing for fact-checking.

While traditional RAG approaches have proven effective, combining RAG with graph learning techniques offers even more powerful capabilities for information retrieval and generation.

Advantages of graph RAG (GRAG) approaches

Graph learning leverages the inherent structure and relationships within data, which can significantly enhance the context and relevance of the information that's retrieved. The general benefits of integrating graphs with RAG are as follows:

- **Structural context**: Graphs capture complex relationships between entities, providing a richer context than flat text documents.
- **Multi-hop reasoning**: Graph structures allow information to be retrieved across multiple connected entities, facilitating more complex reasoning tasks.
- **Improved relevance**: By considering both textual and topological information, graph-based retrieval can identify more pertinent information for the generation process.

In the following sections, we'll explore the advantages of a few specific approaches.

Knowledge graph RAG

Knowledge graph RAG leverages structured knowledge graphs to enhance the retrieval and generation process. This approach offers several key advantages:

- **Precise entity and relationship retrieval**: By utilizing the structured nature of knowledge graphs, this method can retrieve not just relevant entities, but also the relationships between them. For example, a biomedical knowledge graph can retrieve not only a specific drug, but also its interactions with other drugs, its side effects, and its approved uses.
- **Contextual enrichment**: The retrieved information includes the broader context of entities within the graph. This allows the LLM to understand the entity's place in a larger network of information, leading to more nuanced and accurate responses.
- **Hierarchical information access**: Knowledge graphs often contain hierarchical relationships (for example, *is-a* and *part-of*). This structure allows for more flexible retrieval, where the system can access both specific details and broader categories as needed.
- **Multi-hop reasoning**: Knowledge graph RAG can facilitate multi-hop reasoning by retrieving paths of connected entities and relationships. This is particularly useful for complex queries that require information from multiple sources to be pieced together within the graph.

GNNs

As we have seen, GNNs are powerful tools for learning representations of graph-structured data. In the context of RAG, they offer several benefits:

- **Learning graph embeddings**: GNNs can create dense vector representations (embeddings) of nodes, edges, and subgraphs. These embeddings capture both local and global structural information, allowing for more effective retrieval of relevant graph components.

- **Capturing graph topology**: Unlike traditional neural networks, GNNs explicitly model the relationships between entities in a graph. This allows them to capture complex topological features that are crucial for understanding the context and relevance of information in a graph.
- **Scalability**: GNNs can process large-scale graphs efficiently, making them suitable for real-world knowledge bases that often contain millions of entities and relationships.
- **Inductive learning**: Many GNN architectures support inductive learning, allowing them to generalize to unseen nodes or even entirely new graphs. This is particularly useful for dynamic knowledge bases that are constantly updated.

GRAG

Graph RAG (**GRAG**) is an advanced technique that emphasizes the importance of subgraph structures throughout both the retrieval and generation processes:

- **Subgraph-aware retrieval**: Instead of retrieving individual entities or relationships, GRAG focuses on retrieving relevant subgraphs. This approach preserves the local structure around entities of interest, providing a richer context for the generation process.
- **Topology-preserving generation**: GRAG maintains awareness of the graph topology during the generation process. This ensures that the generated text respects the structural relationships present in the retrieved subgraphs, leading to more coherent and factually consistent outputs.
- **Soft pruning**: GRAG often employs a soft pruning mechanism to refine retrieved subgraphs. This process removes less relevant nodes and edges while maintaining the overall structure, helping to focus the LLM on the most pertinent information.
- **Hierarchical text conversion**: GRAG typically includes methods for converting subgraphs into hierarchical text descriptions. This conversion preserves both the textual content and the structural information of the graph, allowing the LLM to work with a rich, structured input.
- **Multi-hop reasoning support**: By maintaining subgraph structures, GRAG naturally supports multi-hop reasoning tasks. The LLM can traverse the retrieved subgraph to connect distant pieces of information, enabling more complex inferencing.

Let's consider how GRAG might be applied in a customer support system for a tech company that utilizes a knowledge base structured as a knowledge graph. The knowledge graph contains interconnected information about products, error codes, troubleshooting steps, and user manuals.

Suppose a user submits the following query: *How do I resolve Error Code E101 on my SmartHome Router?*

Traditional methods might retrieve isolated entities, like the product name or error code, which could lead to fragmented or incomplete answers. GRAG takes a unique approach to provide an accurate and context-rich response:

1. GRAG focuses on subgraph-aware retrieval, extracting a structured, coherent subgraph that relates to *Error Code E101* and the *SmartHome Router*. This subgraph can include details like the cause of the error (e.g., a network configuration conflict), the troubleshooting steps (e.g., updating firmware, resetting the router, or changing network settings), and related links to the router's user manual or firmware update pages.
2. Once the subgraph is retrieved, GRAG applies a soft pruning mechanism to preserve the integrity of the local graph structure. Irrelevant or tangential information, such as troubleshooting steps for other devices or unrelated error codes, is removed.
3. GRAG transforms the pruned subgraph into hierarchical text descriptions, which maintain the graph's structure while converting the information into a form that the LLM can seamlessly process. Our subgraph is converted into a text hierarchy that organizes information as follows:

    I. An overview of *Error Code E101* and its cause (a network configuration conflict)

    II. Troubleshooting steps provided in a logical sequence

    III. Additional details where relevant, including reports from users linking E101 to recent firmware updates

4. GRAG employs topology-preserving generation to ensure that the relationships and structure encoded in the subgraph are reflected in the response. The system can now generate a detailed reply:

    The Error Code E101 on your SmartHome Router typically occurs due to a network configuration conflict. To resolve it, verify if your router's firmware is updated. If it is, reset the router by holding the reset button for 10 seconds. You may also need to update the network settings to avoid IP conflicts. Note that some users have reported encountering this error after recent firmware updates, so rolling back to a previous version might help if the issue persists.

    As mentioned, a key strength of GRAG is its multi-hop reasoning support. For example, the system might connect *Error Code E101* to a network configuration conflict and, from there, to specific troubleshooting steps in the user manual. This enables the LLM to infer complex relationships and provide a comprehensive answer without missing valuable context.

These approaches offer unique strengths in leveraging graph structures for RAG. The choice between them often depends on the specific requirements of the application, the nature of the knowledge base, and the complexity of the queries being handled. Moreover, implementing your chosen approach carefully is crucial because of challenges such as managing large and noisy graphs, maintaining low latency, and adapting LLMs to handle structured graph inputs.

Challenges in integrating LLMs with graph learning

It's worth noting that integrating LLMs with graph learning involves several challenges:

- **Efficiency and scalability**: LLMs require significant computational resources, which poses deployment challenges in real-world applications, particularly on resource-constrained devices. Knowledge distillation, where an LLM (teacher model) transfers knowledge to a smaller, efficient GNN (student model), offers a promising solution.
- **Data leakage and evaluation**: LLMs pre-trained on vast datasets risk data leakage, potentially inflating performance metrics. Mitigating this requires new datasets and careful test data sampling. Establishing fair evaluation benchmarks is also crucial for accurate performance assessment.
- **Transferability and explainability**: Enhancing LLMs' ability to transfer knowledge across diverse graph domains and improving their explainability is vital. Techniques such as chain-of-thought prompting can leverage LLMs' reasoning capabilities for better transparency.
- **Multimodal integration**: Graphs often encompass multiple data types, including images, audio, and numeric data. Extending LLM integration to these multimodal settings represents an exciting research opportunity. With the rapid advancement in the quality of LLM generation, it's going to play a critical role in augmenting intelligence over graph data and learning.

Summary

In this chapter, we explored integrating LLMs with graph learning, highlighting how LLMs can enhance traditional GNNs. We discussed the evolution of LLMs, their capabilities in processing textual data within graphs, and their potential to improve node representations and graph-related tasks. You learned about various approaches for utilizing LLMs in graph learning, including feature-level and text-level enhancements, as well as using LLMs as predictors through techniques such as InstructGLM.

We also presented real-world applications in drug discovery, social network analysis, and financial fraud detection to illustrate the practical benefits of this integration. Furthermore, you became familiar with the challenges in combining LLMs with graph learning, such as computational costs, data bias, and explainability issues, while learning about the potential for future advancements in this field.

In the next chapter, we'll explore applying deep learning to graphs in more depth.

Part 3: Practical Applications and Implementation

In this part of the book, you will dive into the practical implementation of graph deep learning across various domains. You will learn how to apply graph-based approaches to natural language processing, build recommendation systems, and leverage graph structures in computer vision applications.

This part has the following chapters:

7 Graph Deep Learning in Practice

Having explored the theoretical aspects of graph deep learning in the previous chapters, now is a good time to get our hands dirty by diving into its practical applications.

Social networks have become an integral part of our digital lives, generating vast amounts of data that can provide valuable insights into human behavior, relationships, and social dynamics. Graph deep learning offers powerful tools to analyze and extract meaningful information from these complex networks. In this chapter, we will explore practical applications of graph deep learning techniques to social network analysis using **PyTorch Geometric** (**PyG**).

Here, we will focus on a hypothetical dataset representing a social network of university students. This example will demonstrate how graph-based machine learning can be applied to real-world scenarios, such as predicting user interests, recommending new connections, and identifying community structures.

Our dataset consists of the following elements:

- **Nodes**: Each node represents a student in the university.
- **Edges**: Edges between nodes represent friendships or connections between students.
- **Node features**: Each student (node) has associated features, including the following:
    - Age
    - Year of study
    - Academic major (encoded as a one-hot vector)
- **Node labels**: Each student is assigned to one of five interest groups, which we'll use for our node classification task.

Throughout this chapter, we will tackle two main tasks:

- **Node classification**: We'll predict student interests based on their features and connections within the network. This task demonstrates how **graph neural networks** (**GNNs**) can leverage both node attributes and network structure to make predictions about individual nodes.
- **Link prediction**: We'll develop a model to recommend new friendships by predicting potential edges in the graph. This showcases how graph deep learning can be used for recommendation systems and network growth prediction.

We'll implement these tasks using state-of-the-art GNN architectures. Additionally, we'll visualize our results to gain intuitive insights into the network structure and our model's performance.

By the end of this chapter, you'll have a practical understanding of how to apply graph deep learning techniques to social network data, interpret the results, and adapt these methods to your own projects. While we use a synthetic dataset for demonstration, the techniques covered here can be readily applied to real-world social network data.

Let's begin our journey into the practical world of graph deep learning for social network analysis! We will be exploring the following topics at length:

- Setting up the environment
- Creating the graph dataset
- Node classification – predicting student interests
- Link prediction – recommending new friendships

Setting up the environment

Before we dive into the implementation of our graph deep learning models, it's crucial to set up our development environment with the necessary libraries and tools. In this section, we'll cover the essential imports and explain their roles in our project.

First, let's start by importing the required libraries:

```
import torch
import torch_geometric
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv, GAE
from torch_geometric.utils import train_test_split_edges
from torch_geometric.transforms import RandomLinkSplit
from sklearn.manifold import TSNE
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import networkx as nx
from sklearn.metrics import roc_auc_score
```

Let's break down these imports and their purposes:

- `torch`: The core PyTorch library used for tensor operations and building neural networks
- `torch_geometric`: PyG, an extension library to PyTorch for deep learning on graphs and other irregular structures
- `torch_geometric.data.Data`: A class used to create graph data objects in PyG that hold node features, edge indices, and labels
- `torch_geometric.nn.GCNConv`: The **graph convolutional network** (**GCN**) layer, used for node classification
- `torch_geometric.nn.GAE`: The **graph autoencoder** (**GAE**), used for link prediction
- `torch_geometric.utils.train_test_split_edges`: A utility function that helps split the graph data for link prediction tasks
- `torch_geometric.transforms.RandomLinkSplit`: A transform used to prepare the data for link prediction by splitting it into train, validation, and test sets
- `sklearn.manifold.TSNE`: **t-Distributed Stochastic Neighbor Embedding** (**t-SNE**), used for dimensionality reduction when visualizing node embeddings
- `sklearn.cluster.KMeans`: Used for clustering node embeddings to identify social groups
- `matplotlib.pyplot`: A library used for creating visualizations of graphs and results
- `networkx`: A library providing additional tools for working with graphs, used for some visualizations

Make sure you have all these libraries installed in your Python environment. You can install them using `pip`:

```
pip install torch torch_geometric scikit-learn matplotlib networkx
```

Note that the installation of PyG (`torch_geometric`) might require additional steps depending on your system and CUDA version. Please refer to the official documentation for detailed installation instructions: `https://pytorch-geometric.readthedocs.io/en/latest/`.

With these libraries imported, we're now ready to start working with our social network data and implementing our graph deep learning models.

Creating the graph dataset

The next step is to create a synthetic social network dataset to demonstrate our graph deep learning techniques. While real-world data would be ideal, using synthetic data allows us to focus on the implementation and understanding of the algorithms without the complexities of data acquisition and preprocessing.

Here, we will create a social network dataset representing university students. The complete code can be found at `https://github.com/PacktPublishing/Applied-Deep-Learning-on-Graphs`.

Let's break down this code and explain each part.

1. We set a random seed for reproducibility:

   ```
   torch.manual_seed(42)
   ```

   This ensures that we generate the same "random" data each time we run the code.

2. We define our dataset parameters:

   ```
   num_nodes = 1000
   num_features = 10
   num_classes = 5
   ```

   - `num_nodes` is the number of students (nodes) in our network.
   - `num_features` is the number of features for each student. We're using 10 features to represent age, year of study, and 8 possible academic majors.
   - `num_classes` is the number of different interest groups, which will be our target for node classification.

3. We create node features (`x`) using `torch.randn()`:

   ```
   x = torch.randn((num_nodes, num_features))
   ```

   This generates a tensor of size `(num_nodes, num_features)` filled with random values from a standard normal distribution. In a real dataset, these features would be actual attributes of the students.

4. We create edges (`edge_index`) using `torch.randint()`:

   ```
   edge_index = torch.randint(0, num_nodes, (2, 5000))
   ```

   This generates 5,000 random edges between nodes. The `edge_index` tensor has the shape `(2, 5000)`, where each column represents an edge with `[source_node, target_node]`.

5. We create node labels (`y`) using `torch.randint()`:

   ```
   y = torch.randint(0, num_classes, (num_nodes,))
   ```

This assigns each node (student) to one of the five interest groups randomly.

6. We create a PyG `Data` object, which efficiently holds our graph data. We pass in our node features (`x`), edge indices (`edge_index`), and node labels (`y`):

   ```
   data = Data(x=x, edge_index=edge_index, y=y)
   data.num_classes = num_classes
   ```

7. Finally, we print some information about our dataset to verify its structure:

   ```
   print(f"Number of nodes: {data.num_nodes}")
   print(f"Number of edges: {data.num_edges}")
   print(f"Number of node features: {data.num_node_features}")
   print(f"Number of classes: {data.num_classes}")
   ```

This synthetic dataset now represents a social network:

- Each *node* is a student with 10 features (representing attributes such as age, year, and major).
- *Edges* represent friendships between students.
- Each student belongs to one of five interest groups.

The output of the code will be as follows:

```
Number of nodes: 1000
Number of edges: 5000
Number of node features: 10
Number of classes: 5
```

In a real-world scenario, you would typically load your data from files or databases and preprocess it to fit this structure. The `Data` object we've created is now ready to be used with PyG's GNN models.

In the next sections, we'll use this dataset to perform node classification, link prediction, and graph clustering tasks.

Node classification – predicting student interests

In this section, we'll implement a GCN to predict student interests based on their features and connections in the social network. This task demonstrates how GNNs can leverage both node attributes and network structure to make predictions about individual nodes.

Let's start by defining our GCN model:

```
class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
```

```
        self.conv2 = GCNConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index)
        return x

# Initialize the model
model = GCN(in_channels=num_features,
            hidden_channels=16,
            out_channels=num_classes)
```

As you can see, our GCN model consists of two graph convolutional layers. The first layer takes the input features and produces hidden representations, while the second layer produces the final class predictions.

Now, let's train the model. You can view the complete code at `https://github.com/PacktPublishing/Applied-Deep-Learning-on-Graphs`. Let's break down the training process here:

1. We define an optimizer (`Adam`) and a loss function (`CrossEntropyLoss`) for training our model:

   ```
   optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
   criterion = torch.nn.CrossEntropyLoss()
   ```

2. The `train()` function performs one step of training:

   ```
   def train():
       model.train()
       optimizer.zero_grad()
       out = model(data.x, data.edge_index)
       loss = criterion(out, data.y)
       loss.backward()
       optimizer.step()
       return loss
   ```

   - It sets the model to training mode.
   - It computes the forward pass of the model.
   - It calculates the loss between predictions and true labels.
   - It performs backpropagation and updates the model parameters.

3. The `test()` function evaluates the model's performance:

   ```
   def test():
       model.eval()
       out = model(data.x, data.edge_index)
       pred = out.argmax(dim=1)
       correct = (pred == data.y).sum()
       acc = int(correct) / int(data.num_nodes)
       return acc
   ```

   - It sets the model to evaluation mode.
   - It computes the forward pass.
   - It calculates the accuracy of predictions.

4. We train the model for `200` epochs, printing the loss and accuracy every `10` epochs:

   ```
   for epoch in range(200):
       loss = train()
       if epoch % 10 == 0:
           acc = test()
           print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, \
                 Accuracy: {acc:.4f}')
   ```

5. Finally, we evaluate the model one last time to get the final accuracy:

   ```
   final_acc = test()
   print(f"Final Accuracy: {final_acc:.4f}")
   ```

To visualize how well our model is performing, let's create a function to plot the predicted versus true interest groups:

```
def visualize_predictions(model, data):
    model.eval()
    out = model(data.x, data.edge_index)
    pred = out.argmax(dim=1)

    plt.figure(figsize=(10, 5))
    plt.subplot(121)
    plt.title("True Interests")
    plt.scatter(
        data.x[:, 0], data.x[:, 1], c=data.y, cmap='viridis')
    plt.colorbar()

    plt.subplot(122)
    plt.title("Predicted Interests")
```

```
    plt.scatter(
        data.x[:, 0], data.x[:, 1], c=pred, cmap='viridis')
    plt.colorbar()

    plt.tight_layout()
    plt.show()

visualize_predictions(model, data)
```

This function creates two scatter plots: one showing the true interest groups and another showing the predicted interest groups.

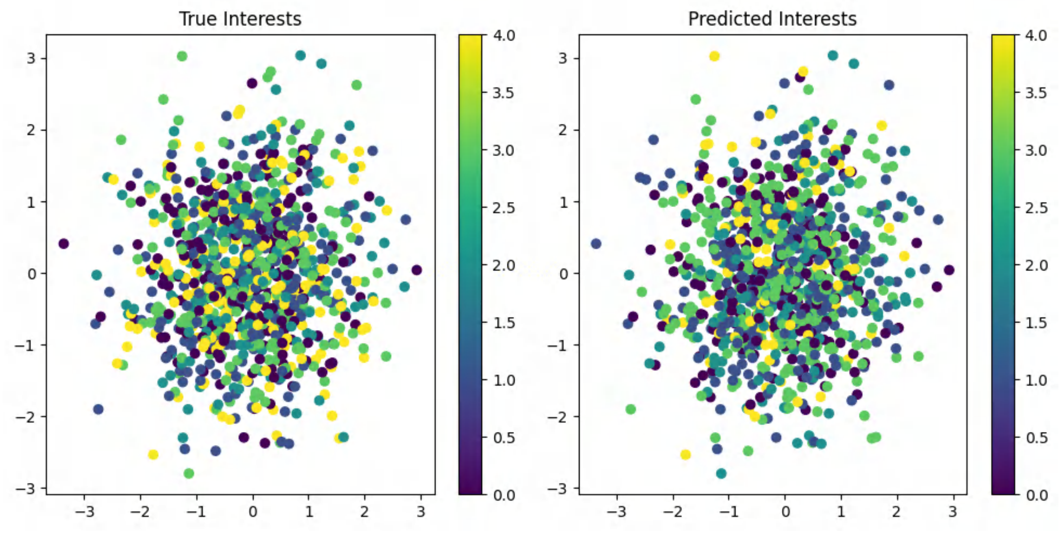


Figure 7.1: Model performance: true versus predicted interests

In *Figure 7.1*, each point represents a student, positioned based on their first two features. The visualization consists of two scatter plots labeled **True Interests** and **Predicted Interests**. Each point in these plots represents one of the students. The colors in both plots, ranging from purple (`0.0`) to yellow (`4.0`), indicate different interest groups. The left plot shows the actual or "true" interest groups of the students, while the right plot displays the model's predictions of these interest groups. The similarity between the distributions conveys the effectiveness of graph learning techniques in such predictions.

In a real-world application, this node classification task could be used to predict student interests based on their profile information and social connections. This could be valuable for personalized content recommendations, targeted advertising, or improving student engagement in university activities.

Remember that while our synthetic dataset provides a clean example, real-world data often requires more preprocessing, handling of missing values, and careful consideration of privacy and ethical concerns when working with personal data.

Another aspect of graph learning is the task of link prediction. This comes up in a lot of real-world scenarios, especially ones where we are trying to predict certain connections.

Link prediction – recommending new friendships

In this section, we'll implement a GAE for link prediction. This task aims to predict potential new connections in the network, which can be used to recommend new friendships among students.

First, let's define our GAE model:

```
class LinkPredictor(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels):
        super().__init__()
        self.encoder = GCNConv(in_channels, hidden_channels)

    def encode(self, x, edge_index):
        return self.encoder(x, edge_index).relu()

    def decode(self, z, edge_label_index):
        return (
            z[edge_label_index[0]] * z[edge_label_index[1]]
        ).sum(dim=-1)

    def forward(self, x, edge_index, edge_label_index):
        z = self.encode(x, edge_index)
        return self.decode(z, edge_label_index)

# Initialize the model
model = LinkPredictor(in_channels=num_features, hidden_channels=64)
```

Our `LinkPredictor` model uses a GCN layer for encoding node features into embeddings, and a simple dot product operation for decoding (predicting links).

Now, let's prepare our data for link prediction using the following code:

```
from torch_geometric.transforms import RandomLinkSplit

# Prepare data for link prediction
transform = RandomLinkSplit(
    num_val=0.1, num_test=0.1, is_undirected=True,
    add_negative_train_samples=False
)
train_data, val_data, test_data = transform(data)

# Define optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

The `RandomLinkSplit` transform splits our graph into training, validation, and test sets for link prediction. It removes some edges for validation and testing and generates negative samples (non-existent edges).

Now, let's define our training and evaluation functions:

1. Initialize the training process, prepare the optimizer, and encode the input data:

```
def train_link_predictor():
    model.train()
    optimizer.zero_grad()
    z = model.encode(train_data.x, train_data.edge_index)
```

2. Now, we define positive edges and generate negative edges for training:

```
    pos_edge_index = train_data.edge_index

    neg_edge_index = torch_geometric.utils.negative_sampling(
        edge_index=pos_edge_index,
        num_nodes=train_data.num_nodes,
        num_neg_samples=pos_edge_index.size(1),
    )
```

3. Combine positive and negative edges and create corresponding labels:

```
    edge_label_index = torch.cat([
        pos_edge_index, neg_edge_index], dim=-1)
    edge_label = torch.cat([
        torch.ones(pos_edge_index.size(1)),
        torch.zeros(neg_edge_index.size(1))
    ], dim=0)
```

4. The model makes predictions, calculates the loss, performs backpropagation, and updates the model parameters:

```
    out = model.decode(z, edge_label_index)
    loss = torch.nn.BCEWithLogitsLoss()(out, edge_label)

    loss.backward()
    optimizer.step()
    return loss
```

5. This begins the testing function, setting the model to evaluation mode and making predictions without gradient calculation:

```
def test_link_predictor(data):
    model.eval()
    with torch.no_grad():
        z = model.encode(data.x, data.edge_index)
        out = model.decode(z, data.edge_label_index)
```

6. Finally, we convert the predictions to NumPy arrays and calculate the **Area Under the Receiver Operating Characteristic Curve** (**ROC AUC)** score to evaluate the model's performance:

```
    y_true = data.edge_label.cpu().numpy()
    y_pred = out.cpu().numpy()

    return roc_auc_score(y_true, y_pred)
```

Now, let's write the training and prediction modules:

1. We train the model for `100` epochs, printing the loss and validation AUC every `10` epochs:

```
for epoch in range(100):
    loss = train_link_predictor()
    if epoch % 10 == 0:
```

   The `train_link_predictor()` function encodes the node features, predicts links for both positive and negative samples, and computes the binary cross-entropy loss.

2. The `test_link_predictor()` function evaluates the model's performance using the AUC metric:

```
        val_auc = test_link_predictor(val_data)
        print(f'Epoch: {epoch:03d}, Loss: {loss:.4f}, \
              Val AUC: {val_auc:.4f}')
```

3. Finally, we evaluate the model on the test set to get the final AUC score:

```
test_auc = test_link_predictor(test_data)
print(f"Test AUC: {test_auc:.4f}")
```

   Here is the code output:

```
Epoch: 000, Loss: 0.6911, Val AUC: 0.5528
Epoch: 010, Loss: 0.6779, Val AUC: 0.5297
Epoch: 020, Loss: 0.6732, Val AUC: 0.5068
Epoch: 030, Loss: 0.6696, Val AUC: 0.5156
Epoch: 040, Loss: 0.6666, Val AUC: 0.5172
Epoch: 050, Loss: 0.6668, Val AUC: 0.5139
```

```
Epoch: 060, Loss: 0.6642, Val AUC: 0.5142
Epoch: 070, Loss: 0.6631, Val AUC: 0.5132
Epoch: 080, Loss: 0.6611, Val AUC: 0.5130
Epoch: 090, Loss: 0.6600, Val AUC: 0.5135
Test AUC: 0.4843
```

To demonstrate how we can use this model to recommend new friendships, let's create a function that predicts the most likely new connections for a given student:

1. Define the function, set the model to evaluation mode, and encode the graph data:

```
def recommend_friends(student_id, top_k=5):
    model.eval()
    with torch.no_grad():
        z = model.encode(data.x, data.edge_index)
```

2. Here, we create an edge index for all possible connections between the given student and other students:

```
        other_students = torch.arange(data.num_nodes)
        other_students = other_students[
            other_students != student_id]
        edge_index = torch.stack([
            torch.full_like(other_students, student_id),
            other_students
        ])
```

3. This code predicts scores for all possible connections and selects the top *k* recommendations based on these scores:

```
        scores = model.decode(z, edge_index)

        top_scores, top_indices = scores.topk(top_k)
        recommended_friends = other_students[top_indices]
```

4. Finally, the function returns the recommended friends and their corresponding scores, as you can see in the following example:

```
    return recommended_friends, top_scores

# Example usage
student_id = 42  # Example student ID
recommended_friends, scores = recommend_friends(student_id)
print(f"Top 5 friend recommendations for student {student_id}:")
for friend, score in zip(recommended_friends, scores):
    print(f"Student {friend.item()}: Score {score:.4f}")
```

The code output is as follows:

```
Top 5 friend recommendations for student 42:
Student 381: Score 1.8088
Student 91: Score 1.6567
Student 662: Score 1.5878
Student 467: Score 1.5143
Student 870: Score 1.4449
```

This `recommend_friends` function takes a student ID and returns the top *k* recommended new connections based on the link prediction model.

In a real-world application, this link prediction model could be used to power a friend recommendation system in a university's social network platform. It could help students expand their social circles based on shared characteristics and network structure.

The link prediction model's applications extend far beyond just university social networks, encompassing diverse domains such as professional networking platforms such as LinkedIn for career connections and business collaborations, academic research networks for finding potential co-authors and research partners, business applications for supply chain partner matching and B2B networking, healthcare networks for patient referrals and provider collaborations, community building through interest-based group recommendations and event participant matching, and content recommendation systems for discovering similar resources and expert-content matching. These applications demonstrate the model's versatility in enhancing connectivity and collaboration opportunities across various sectors and use cases.

Remember that when implementing such systems with real user data, it's crucial to consider privacy implications and potentially incorporate additional factors (such as user preferences or mutual friends) into the recommendation algorithm.

Summary

In this chapter, we covered the practical applications of graph deep learning in social network analysis using PyG. We focused on a hypothetical dataset representing a university student social network, demonstrating how graph-based machine learning can be applied to real-world scenarios.

Together, we achieved two main tasks: node classification for predicting user interests and link prediction for recommending new connections. By following step-by-step instructions, you learned how to create a synthetic dataset, implement GCNs for node classification, and use GAEs for link prediction. We broke down our code into snippets in relation to data preparation, model training, evaluation, and visualization, allowing you to understand the practical aspects of applying graph deep learning techniques to social network data.

In the upcoming chapters, we will explore how graph deep learning is being applied to various domains, starting with natural language processing.

8
Graph Deep Learning for Natural Language Processing

Language, by its very nature, is inherently structured and relational. Words form sentences, and sentences form documents, which contain concepts that interlink in complex ways to convey meaning. Graph structures provide an ideal framework to capture these intricate relationships, going beyond the traditional models. By representing text as graphs, we can leverage the rich expressiveness of graph theory and the computational power of deep learning to tackle challenging **natural language processing** (**NLP**) problems.

In this chapter, we will delve into the fundamental concepts of graph representations in NLP, exploring various types of linguistic graphs such as dependency trees, co-occurrence networks, and knowledge graphs. We'll then build upon this foundation to examine the architectures and mechanisms of **graph neural networks** (**GNNs**) that have been specifically adapted for language tasks.

We'll cover the following topics:

- Graph structures in NLP
- Graph-based text summarization
- **Information extraction** (**IE**) using GNNs
- Graph-based **dialogue systems**

Graph structures in NLP

NLP has seen significant advancements in recent years, with graph-based approaches emerging as a powerful paradigm for representing and processing linguistic information. In this section, we introduce the concept of graph structures in NLP, highlighting their importance and exploring various types of linguistic graphs.

Importance of graph representations in language

Graph representations play a crucial role in capturing the inherent structure and relationships within language. They offer several advantages over traditional sequential or **bag-of-word** modeling, which is a simple text representation technique that converts a document into a vector by counting the frequency of each word, disregarding grammar and word order:

- **Structural information**: Graphs can explicitly represent the hierarchical and relational nature of language, preserving important linguistic structures that may be lost in linear representations.
- **Contextual relationships**: By connecting related elements, graphs capture long-range dependencies and contextual information more effectively than sequential models.
- **Flexibility**: Graph structures can represent various levels of linguistic information, from word-level relationships to document-level connections and even cross-document links.
- **Interpretability**: Graph representations often align with human intuition about language structure, making them more interpretable and analyzable.
- **Integration of external knowledge**: Graphs facilitate the incorporation of external knowledge sources, such as ontologies or knowledge bases, into NLP models.
- **Multi-modal integration**: Graph structures can naturally represent relationships between different modalities, such as text, images, and speech.

Types of linguistic graphs

Several types of graph structures are commonly used in NLP, each capturing different aspects of linguistic information:

- **Dependency trees** represent grammatical relationships between words in a sentence:
  - **Nodes** are words and **edges** indicate syntactic dependencies.
  - **Example**: In "*The cat chased the mouse*," "*chased*" would be the root, with "*cat*" and "*mouse*" as its dependents. *Figure 8.1* illustrates a dependency tree representation of this sentence and shows how dependency trees represent grammatical relationships between words in a sentence:

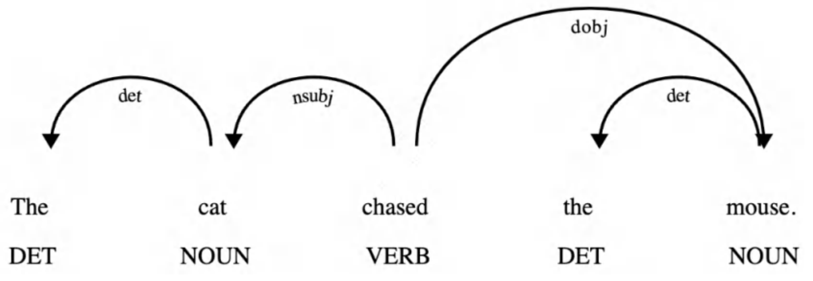


Figure 8.1 – Dependency tree representation of the sentence "The cat chased the mouse"

The diagram depicts "*chased*" as the root verb, with arrows indicating syntactic dependencies to other words such as "*cat*" (subject) and "*mouse*" (object). It also includes part-of-speech tags for each word, such as **DET** (determiner), **NOUN**, and **VERB**.

- **Co-occurrence graphs** capture word associations based on their co-occurrence in a corpus. They are useful for tasks such as word sense disambiguation and semantic similarity.
  - **Nodes** represent words and **edges** indicate how often they appear together.
  - **Example**: Let's say we have the following sentences:
    - "*The cat and dog play.*"
    - "*The dog chases the cat.*"
    - "*The cat sleeps on the mat.*"

  In this example, we'll create a co-occurrence graph where words that appear in the same sentence are connected:

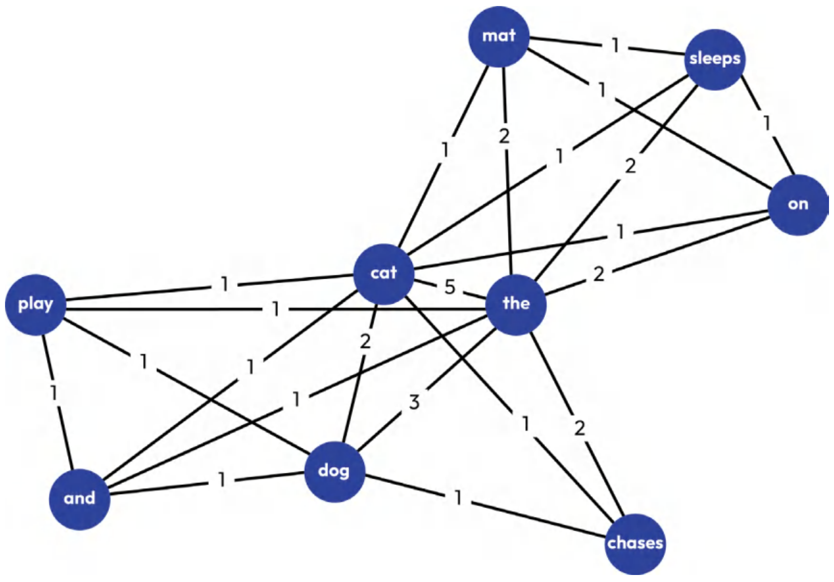


Figure 8.2 – Word co-occurrence graph

- **Knowledge graphs** represent factual information and relationships between entities:
  - **Nodes** are entities (e.g., people, places, concepts) and **edges** are relationships.
  - **Example**: Some examples include ConceptNet, Wikidata, and domain-specific ontologies.

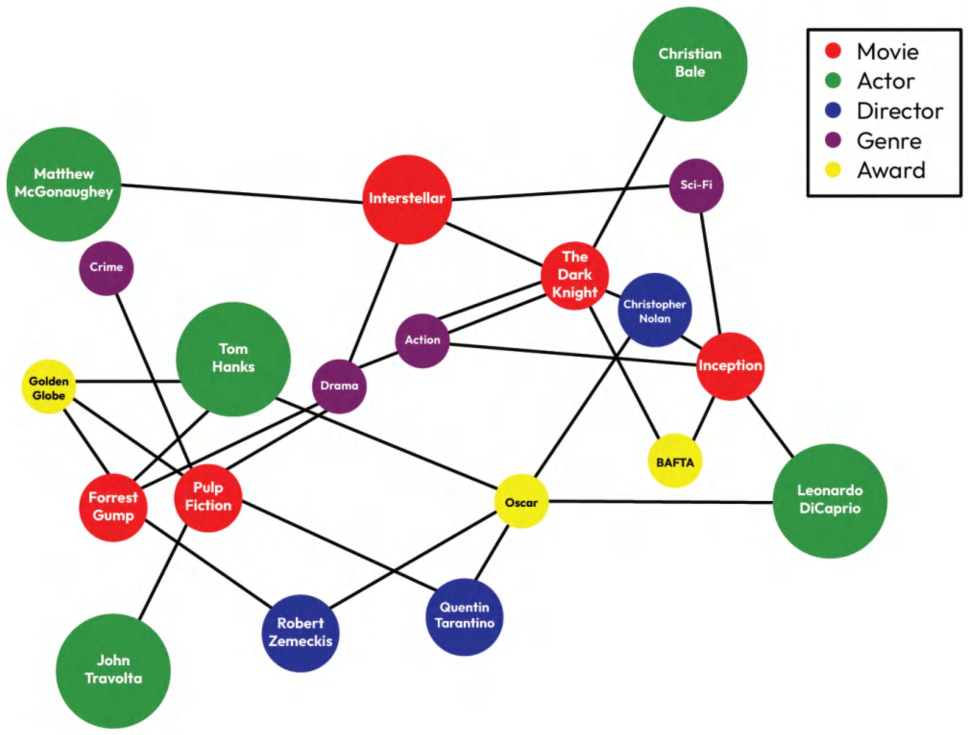


Figure 8.3 – Movie industry knowledge graph

Figure 8.3 illustrates relationships between various elements of the film industry, including directors, movies, genres, and awards. The *nodes* represent entities such as filmmakers (e.g., **Christopher Nolan**, **Quentin Tarantino**), films (e.g., **Interstellar**, **Pulp Fiction**), genres (e.g., **sci-fi**, **crime**), and awards (e.g., **Oscar**, **BAFTA**). The connections between these nodes demonstrate the complex interplay of creative talent, film categories, and industry recognition, offering insights into the multifaceted nature of cinema and its key players.

- **Semantic graphs** represent the meaning of sentences or documents. They are used in tasks such as semantic parsing and abstract meaning representation. **Semantic parsing** in NLP is the task of converting natural language expressions into formal, structured representations of their meaning. It involves mapping words and phrases to concepts, identifying relationships, and generating logical forms or executable code that capture the underlying semantics of the input text. This process enables machines to understand and act upon human language in a more precise and actionable manner.
    - **Nodes** can be concepts, events, or propositions, with **edges** showing semantic relationships.
    - **Example**: The semantic graph in *Figure 8.4* illustrates how the sentence "*John read a book in the library*" can be broken down into its constituent parts and relationships. The graph consists of nodes representing concepts and edges showing the semantic relationships between them:

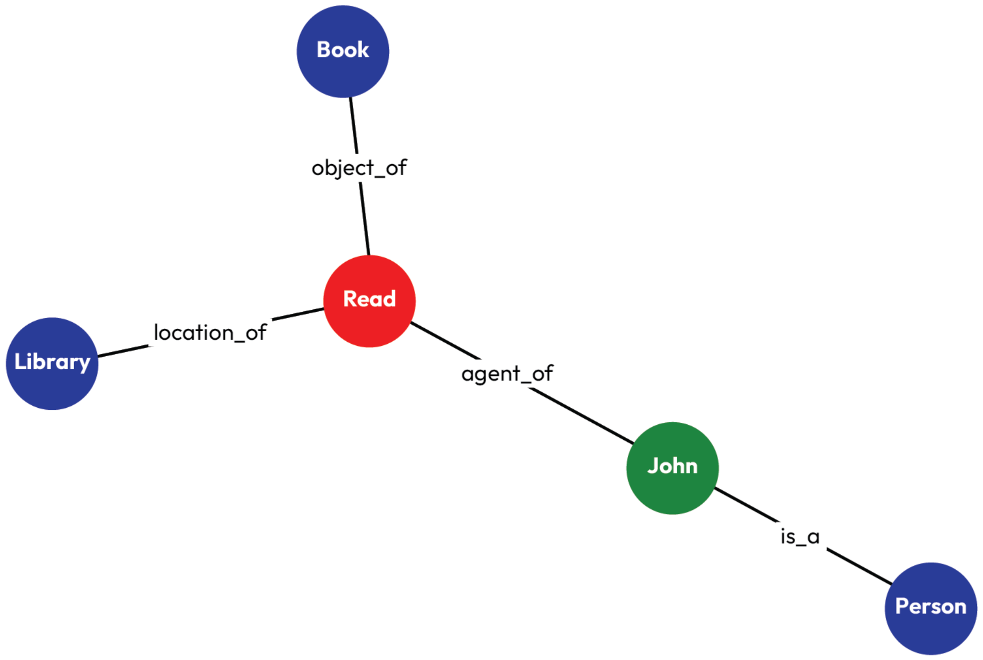


Figure 8.4 – Semantic graphical representation

Specifically, the graph shows the following:

- "*John*" is connected to "*Read*" with the label **agent_of**, indicating John is the one performing the action.
- "*Book*" is connected to "*Read*" with the label **object_of**, showing that the book is what's being read.
- "*Library*" is connected to "*Read*" with the label **location_of**, indicating where the reading took place.
- "*John*" is also connected to "*Person*" with an **is_a** relationship, classifying John as a person.

- **Discourse graphs** represent the structure of longer texts or conversations and are used in tasks such as dialogue understanding and text coherence analysis.
  - **Nodes** can be sentences or utterances, with **edges** showing discourse relations.

- **Example**: *Figure 8.5* depicts a discourse graph representing a simple conversation. The graph consists of seven nodes (**U0** to **U6**), each representing an utterance in the conversation, connected by directed edges that show the flow and relationships between the utterances. The edges are labeled with discourse relations such as **greeting-response**, **acknowledgment**, **question**, **response**, and **acknowledgment-farewell**. The conversation begins with a greeting (**U0**), followed by a response (**U1**), which then branches out to an acknowledgment (**U2**) and a question (**U3**). The dialogue continues with further responses and concludes with a farewell (**U6**). The legend provides brief snippets of each utterance, giving context to the conversation flow:

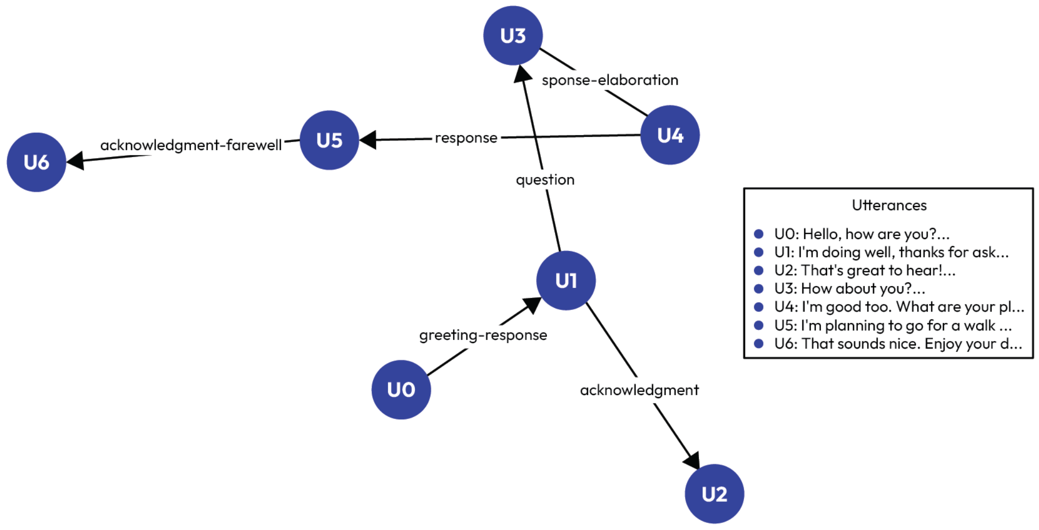

U3
sponse-elaboration
U4
response
U5
acknowledgment-farewell
U6
question
U1
greeting-response
U0
acknowledgment
U2
Utterances
● U0: Hello, how are you?...
● U1: I'm doing well, thanks for ask...
● U2: That's great to hear!...
● U3: How about you?...
● U4: I'm good too. What are your pl...
● U5: I'm planning to go for a walk ...
● U6: That sounds nice. Enjoy your d...

Figure 8.5 – Discourse graph: simple conversation

This visual representation effectively illustrates how discourse graphs can be used to analyze the structure, coherence, and progression of a conversation, making it a valuable tool for tasks such as dialogue understanding and text coherence analysis.

Now, let's examine a few real-world use cases of graph learning in NLP.

Graph-based text summarization

Graph-based approaches have become increasingly popular in text summarization due to their ability to capture complex relationships between textual elements. Here, we will explore two main categories of graph-based summarization: extractive and abstractive.

Extractive summarization using graph centrality

Extractive summarization involves selecting and arranging the most important sentences from the original text to form a concise summary. Graph-based methods for extractive summarization typically follow these steps:

1. Construct a graph representation of the text.
2. Apply centrality measures to identify important nodes (sentences).
3. Extract top-ranked sentences to form the summary.

Graph construction

The text is represented as a graph where *nodes* are sentences and *edges* represent similarities between sentences. Common similarity measures include the following:

- Cosine similarity of **term frequency-inverse document frequency** (**TF-IDF**) vectors
- **Jaccard similarity** of word sets
- **Semantic similarity** using word embeddings

Centrality measures

Several **graph centrality** measures can be used to rank the importance of sentences:

- **Degree centrality** measures the number of connections a node has.
- **Eigenvector centrality** considers the importance of neighboring nodes.
- **PageRank** is a variant of eigenvector centrality, originally used by Google for ranking web pages.
- **Hyperlink-Induced Topic Search** (**HITS**) computes hub and authority scores for nodes.

Example – TextRank algorithm

TextRank, proposed by Mihalcea and Tarau in 2004 (`https://aclanthology.org/W04-3252/`), is a popular graph-based extractive summarization method inspired by PageRank.

Let's look at a simplified Python implementation:

1. First, we import the necessary libraries – `NetworkX` for graph operations, `NumPy` for numerical computations, and `scikit-learn` for TF-IDF vectorization and cosine similarity calculation:

```
import networkx as nx
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer
```

2. Then, we define the `textrank` function, which takes a list of sentences and the number of top sentences to return. It creates a TF-IDF matrix from the sentences and computes a similarity matrix using cosine similarity:

```
def textrank(sentences, top_n=2):
    tfidf = TfidfVectorizer().fit_transform(sentences)
    similarity_matrix = cosine_similarity(tfidf)
```

3. We create a graph from the similarity matrix and compute `pagerank` on this graph. Each sentence is a *node*, and the similarities are *edge weights*:

```
    graph = nx.from_numpy_array(similarity_matrix)
    scores = nx.pagerank(graph)
```

4. Finally, we sort the sentences based on their `pagerank` scores and return the top `n` sentences as the summary:

```
    ranked_sentences = sorted(((
        scores[i], s) for i, s in enumerate(sentences)
    ), reverse=True)
    return [s for _, s in ranked_sentences[:top_n]]
```

Here is an example usage of the `textrank` function. In this case, we define a sample text, split it into sentences, apply the `textrank` algorithm, and print the summary:

```
text = """
Natural language processing (NLP) is a subfield of linguistics,
computer science, and artificial intelligence concerned with the
interactions between computers and human language. The ultimate
objective of NLP is to read, decipher, understand, and make sense of
human languages in a valuable way. NLP is used in many applications,
including machine translation, speech recognition, and chatbots.
"""
sentences = text.strip().split('.')
summary = textrank(sentences)
print("Summary:", ' '.join(summary))
```

Abstractive summarization with graph-to-sequence models

Abstractive summarization aims to generate new sentences that capture the essence of the original text. Graph-to-sequence models have shown promising results in this area by leveraging the structural information of the input text.

For abstractive summarization, graphs are often constructed to represent more fine-grained relationships:

- **Nodes**: Words, phrases, or concepts
- **Edges**: Syntactic dependencies, semantic relations, or co-occurrence

A typical graph-to-sequence model for abstractive summarization consists of the following:

- A **graph encoder** uses GNNs (e.g., **graph convolutional networks** (**GCNs**) or **graph attention network** (**GATs**)) to encode the input graph.
- A **sequence decoder** generates the summary text, often using attention mechanisms to focus on relevant parts of the encoded graph.

Abstract meaning representation (AMR)-to-text summarization

Abstract meaning representation (**AMR**) is a semantic graph representation of text. **AMR-to-text summarization** is an example of graph-to-sequence abstractive summarization. Let's consider a conceptual example using `PyTorch`:

1. We import the necessary libraries – `PyTorch` for deep learning operations, `PyTorch Geometric` for GNNs, and specific modules for neural network layers and functions:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.data import Data
```

2. Then, we define the `AMRToTextSummarizer` class, which is a neural network module. It initializes a GCN layer, a **gated recurrent unit** (**GRU**) layer, and a fully connected layer:

```
class AMRToTextSummarizer(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.graph_conv = GCNConv(input_dim, hidden_dim)
        self.gru = nn.GRU(
            hidden_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
```

3. This is the forward pass of the network. We first apply graph convolution to encode the graph structure, then use a GRU for sequence decoding, and finally apply a fully connected layer to produce the output:

```
    def forward(self, data):
        # Graph encoding
        x, edge_index = data.x, data.edge_index
        h = self.graph_conv(x, edge_index)
        h = F.relu(h)

        # Sequence decoding
        h = h.unsqueeze(0)  # Add batch dimension
        output, _ = self.gru(h)
        output = self.fc(output)

        return output
```

4. We set up the model parameters and create a sample graph for demonstration. Here, we define the dimensions for the input, hidden layer, and output and create a graph with three nodes:

```
input_dim = 100  # Dimension of input node features
hidden_dim = 256
output_dim = 10000  # Vocabulary size

edge_index = torch.tensor([
    [0, 1, 2], [1, 2, 0]], dtype=torch.long)
x = torch.randn(3, input_dim)

data = Data(x=x, edge_index=edge_index)
```

5. Finally, we instantiate the model, run a forward pass with the sample data, and print the shape of the output logits. The output shape would be (1, 3, 10000), representing a batch size of 1, 3 nodes, and logits over a vocabulary of 10,000 words:

```
model = AMRToTextSummarizer(input_dim, hidden_dim, output_dim)
summary_logits = model(data)

print("Summary logits shape:", summary_logits.shape)
```

This example demonstrates the basic structure of a graph-to-sequence model for abstractive summarization. In practice, more sophisticated architectures, attention mechanisms, and training procedures would be employed.

Information extraction (IE) using GNNs

IE is a crucial task in NLP that involves automatically extracting structured information from unstructured text. GNNs have shown promising results in this domain, particularly in event extraction and open IE. In this section, we explore how GNNs are applied to these IE tasks.

Event extraction

Event extraction is the task of identifying and categorizing events mentioned in text, along with their participants and attributes. GNNs have proven effective in this task due to their ability to capture complex relationships between entities and events in a document.

Graph construction for event extraction

In event extraction, a document is typically represented as a graph where **nodes** represent entities, events, and tokens, while **edges** represent various relationships such as syntactic dependencies, coreference links, and temporal order.

Consider the following sentence: "*John Smith resigned as CEO of TechCorp on Monday*."

The graph representation might include the following:

- **Nodes**: John Smith (person), TechCorp (organization), CEO (role), Monday (time), Resignation (event)
- **Edges**: (John Smith) -[AGENT]-> (Resignation), (Resignation) -[ROLE]-> (CEO), (Resignation) -[ORG]-> (TechCorp), (Resignation) -[TIME]-> (Monday)

GNN-based event extraction models

GNN models for event extraction typically involve the following:

1. Encoding the initial node features using pre-trained word embeddings or contextual embeddings.
2. Applying multiple layers of graph convolution to propagate information across the graph.
3. Using the final node representations to classify events and their arguments.

Figure 8.6 shows an example architecture:

```
Input Text -> Graph Construction -> Node Feature Initialization ->
Graph Convolutional Layers -> Event Classification -> Agument Role Labeling
```

Figure 8.6 – GNN-based event extraction model architecture and process flow

A recent study (`https://aclanthology.org/2021.acl-long.274/`) has shown that GNN-based models can outperform traditional sequence-based models, especially in capturing long-range dependencies and handling multiple events in a single document. Specifically, graph-based methods showed significant improvements in handling cross-sentence events and multiple event scenarios through a heterogeneous graph interaction network that captured global context and a *Tracker* module that modeled interdependencies between events. The model's effectiveness was particularly evident when extracting events that involved many scattered arguments across different sentences, validating GNNs' superior ability to capture long-range dependencies compared to traditional sequence models.

Open IE

Open information extraction (**OpenIE**) aims to extract relational triples (subject, relation, object) from text without being confined to a predefined set of relations. GNNs have been successfully applied to this task, leveraging the inherent graph-like structure of sentences.

Graph-based OpenIE approach

In a graph-based OpenIE system, sentences are typically converted into dependency parse trees, which are then used as the input graph for a GNN.

For example, in the sentence "*Einstein developed the theory of relativity*," the dependency parse might look like the following:

```
Einstein --nsubj--> developed <--dobj-- theory
                                          |
                                        nmod
                                          |
                                     relativity
```

Figure 8.7 – Graph-based OpenIE approach using dependency parsing

GNN processing for OpenIE

The GNN processes this graph in several steps:

1. **Node encoding**: Each word is encoded using contextual embeddings.
2. **Graph convolution**: Information is propagated along the dependency edges.
3. **Relation prediction**: The model predicts potential relations between pairs of nodes.
4. **Triple formation**: Valid subject-relation-object triples are constructed based on the predictions.

Using our previous example, the output would look like this: (Einstein, developed, theory of relativity).

Advantages of GNN-based IE

GNN-based approaches to IE offer several advantages:

- Capturing long-range dependencies that may be missed by sequential models
- Integrating various types of linguistic information (syntactic, semantic, coreference) into a unified framework
- Handling documents with complex structures, such as scientific papers or legal documents

Future research in this area may focus on combining GNNs with other deep learning architectures, such as transformers, to create hybrid models that leverage the strengths of both approaches for more accurate and comprehensive information extraction.

Graph-based dialogue systems

Dialogue systems are sophisticated AI-powered applications designed to facilitate human-computer interaction through natural language. These systems employ various NLP techniques such as natural language understanding, dialogue management, and natural language generation to interpret user inputs, maintain context, and produce appropriate responses. Modern dialogue systems often integrate machine learning algorithms to improve their performance over time, adapting to user preferences and learning from past interactions. They find applications in diverse fields, including customer service, virtual assistants, educational tools, and interactive storytelling, continuously evolving to provide more natural and effective communication between humans and machines.

Graph-based approaches have shown significant promise in enhancing the performance and capabilities of dialogue systems. By leveraging graph structures to represent dialogue context, knowledge, and semantic relationships, these systems can better understand user intents, track conversation states, and generate more coherent and contextually appropriate responses.

Dialogue state tracking with GNNs

Dialogue state tracking (**DST**) is a crucial component of task-oriented dialogue systems, responsible for maintaining an up-to-date representation of the user's goals and preferences throughout the conversation. GNNs have been successfully applied to improve the accuracy and robustness of DST.

In a typical GNN-based DST approach, the dialogue history is represented as a graph, where *nodes* represent utterances, slots, and values, while *edges* capture the relationships between these elements. As the conversation progresses, the graph is dynamically updated, and GNN layers are applied to propagate information across the graph, enabling more accurate state predictions.

For example, the **graph state tracker** (**GST**) proposed by Chen et al. in 2020 (`https://doi.org/10.1609/aaai.v34i05.6250`) uses a GAT to model the dependencies between different dialogue elements. This approach has shown superior performance on benchmark datasets such as MultiWOZ, particularly in handling complex multi-domain conversations.

Graph-enhanced response generation

Graph structures can also significantly improve the quality and relevance of generated responses in both task-oriented and open-domain dialogue systems. By incorporating knowledge graphs or conversation flow graphs, these systems can produce more informative, coherent, and contextually appropriate responses.

One approach is to use graph-to-sequence models, where the input dialogue context is first converted into a graph representation, and then a graph-aware decoder generates the response. This allows the model to capture long-range dependencies and complex relationships within the conversation history.

For instance, the **GraphDialog** model introduced by Yang et al. in 2021 (`https://doi.org/10.18653/v1/2020.emnlp-main.147`) constructs a dialogue graph that captures both the local context (recent utterances) and global context (overall conversation flow). The model then uses graph attention mechanisms to generate responses that are more consistent with the entire conversation history. This approach represents conversations as structured graphs where *nodes* represent utterances and *edges* capture various types of relationships between them, such as temporal sequence and semantic similarity. The graph structure allows the model to better understand long-range dependencies and thematic connections across the dialogue, moving beyond the limitations of traditional sequential models. Furthermore, the graph attention mechanism helps the model focus on relevant historical context when generating responses, even if it occurred many turns earlier in the conversation.

This architecture has shown particular effectiveness in maintaining coherence during extended conversations and handling complex multi-topic dialogues where context from different parts of the conversation needs to be integrated.

Knowledge-grounded conversations using graphs

Incorporating external knowledge into dialogue systems is crucial for generating informative and engaging responses. Graph-based approaches offer an effective way to represent and utilize large-scale knowledge bases in conversation models.

Knowledge graphs can be integrated into dialogue systems in several ways:

- **As a source of factual information**: The system can query the knowledge graph to retrieve relevant facts and incorporate them into responses.
- **For entity linking and disambiguation**: Graph structures can help resolve ambiguities and link mentions in the conversation to specific entities in the knowledge base.
- **To guide response generation**: The graph structure can inform the generation process, ensuring that the produced responses are consistent with the known facts and relationships.

An example of this approach is the **Knowledge-Aware Graph-Enhanced GPT-2** (**KG-GPT2**) model proposed by W Lin et al. (`https://doi.org/10.48550/arXiv.2104.04466`). This model incorporates a knowledge graph into a pre-trained language model, allowing it to generate more informative and factually correct responses in open-domain conversations.

Imagine you're using a virtual assistant to plan a trip to London. You start by asking about hotels, then restaurants, and finally transportation. A traditional GPT-2-based system might struggle to connect related information across these different domains. For instance, if you mention wanting a "*luxury hotel in central London*" and later ask about "*restaurants near my hotel*," the system needs to understand that you're looking for high-end restaurants in central London.

The proposed model in the aforementioned paper solves this by using graph networks to create connections between related information. It works in three steps:

1. First, it processes your conversation using GPT-2 to understand the context.
2. Then, it uses GATs to connect related information (such as location, price range, etc.) across different services.
3. Finally, it uses this enhanced understanding to make better predictions about what you want.

The researchers found this approach particularly effective when dealing with limited training data. In real terms, this means the system could learn to make good recommendations even if it hasn't seen many similar conversations before. For example, if it learns that people booking luxury hotels typically also book high-end restaurants and premium taxis, it can apply this pattern to new conversations.

Their approach showed significant improvements over existing systems, especially in understanding relationships between different services (such as hotels and restaurants) and maintaining consistency throughout the conversation. This makes the system more natural and efficient for real-world applications such as travel booking or restaurant reservation systems.

Graph-based dialogue policy learning

In task-oriented dialogue systems, graph structures can also be leveraged to improve dialogue policy learning. By representing the dialogue state, action space, and task structure as a graph, reinforcement-learning algorithms can more effectively explore and exploit the action space.

For example, the **Graph-Based Dialogue Policy** (**GDP**) framework introduced by Chen et al. in 2021 (`https://aclanthology.org/C18-1107`) uses a GNN to model the relationships between different dialogue states and actions. This approach enables more efficient policy learning, especially in complex multi-domain scenarios.

There is an underlying scalability problem with using graphs for language understanding related to nonlinear memory complexity. The quadratic memory complexity issue in graph-based NLP arises because when converting text into a fully connected graph, each token/word needs to be connected to every other token, resulting in $O(n^2)$ connections where n is the sequence length.

For example, in a 1,000-word document, 1 million edges must be stored in memory. This becomes particularly problematic with transformer-like architectures where each connection also stores attention weights and edge features. Modern NLP tasks often deal with much longer sequences or multiple documents simultaneously, making this quadratic scaling unsustainable for both memory usage and computational resources. Common mitigation strategies include sparse attention mechanisms, hierarchical graph structures, and sliding window approaches, but these can potentially lose important long-range dependencies in the text. Please refer to *Chapter 5* for a more in-depth discussion of approaches to the issue of scalability.

Summary

In this chapter, we covered a wide range of topics, starting with the fundamental concepts of graph representations in NLP and progressing through various applications. These applications include graph-based text summarization, IE using GNNs, OpenIE, mapping natural language to logic, question answering over knowledge graphs, and graph-based dialogue systems.

You learned that graph-based approaches offer powerful tools for enhancing various aspects of dialogue systems, from state tracking to response generation and policy learning. As research in this area continues to advance, we can expect to see even more sophisticated and capable dialogue systems that leverage the rich structural information provided by graph representations.

In the next chapter, we will go through some of the very common use cases of graph learning around recommendation systems.

9 Building Recommendation Systems Using Graph Deep Learning

Recommendation systems have become an integral part of our digital landscape, profoundly shaping how we interact with content, products, and services across various industries. From e-commerce giants such as Amazon to streaming platforms such as Netflix, and social media networks such as Facebook, these systems play a crucial role in enhancing user experience, driving engagement, and boosting business outcomes. As we delve into the world of building recommendation systems using graph deep learning, it's essential to understand the evolution of these systems and the transformative potential of graph-based approaches.

Traditionally, recommendation systems have relied on techniques such as **collaborative filtering** (**CF**), content-based filtering, and hybrid approaches. While these methods have been successful to a certain extent, they often fall short in capturing the complex, interconnected nature of user-item interactions and the rich contextual information surrounding these interactions. This is where graph deep learning enters the picture, offering a paradigm shift in how we approach recommendation tasks.

Graph-based methods in recommendation systems offer numerous advantages, including providing a rich multi-modal representation of diverse information types, the ability to capture higher-order relationships beyond direct user-item interactions, handle data sparsity effectively through information propagation, inductive learning capabilities for addressing cold start problems, enhanced interpretability through graph structure analysis, and flexibility in incorporating various types of side information and heterogeneous data. Collectively, these features enable graph-based approaches to provide more comprehensive, accurate, and adaptable recommendations by leveraging the complex relationships and structures inherent in user-item interaction data and associated contextual information.

As we progress through this chapter, we'll explore the following topics:

- Fundamentals of recommendation systems
- Graph structures in recommendation systems
- Graph-based recommendation models
- Training graph deep learning models
- Explainability in graph-based recommendations
- The **cold start problem**

Fundamentals of recommendation systems

Recommendation systems, also known as **recommender systems**, are intelligent algorithms that are designed to predict and suggest items or content that users might find interesting or relevant. These systems analyze patterns in user behavior, preferences, and item characteristics to generate personalized recommendations. The primary purpose of recommendation systems is to enhance the user experience by providing relevant content, increasing user engagement and retention, driving sales and conversions in e-commerce platforms, facilitating content discovery in large item catalogs, and personalizing services across various domains. Recommendation systems play a crucial role in addressing the information overload problem by filtering and prioritizing content based on user preferences and behavior.

Recommendation systems have become an integral part of our digital experiences, influencing our choices in various domains, such as e-commerce, entertainment, social media, and more. This section delves into the core concepts, types, and evaluation metrics of recommendation systems, providing a solid foundation for understanding their role in modern applications.

Let's consider a streaming service that's trying to build a recommendation system for its users. The company has user-movie interaction data as well as metadata for every single movie available. We'll look at how we can leverage this data to provide better recommendations and increase the net promoter scores and watch times of the users effectively.

Types of recommendation systems

There are three main types of recommendation systems, each with its unique approach to generating recommendations:

- **CF** is based on the premise that users with similar preferences in the past will have similar preferences in the future. It utilizes user-item interaction data to identify patterns and make predictions. There are two main approaches: user-based CF, which recommends items liked by similar users, and item-based CF, which recommends items similar to those the user has liked in the past. CF has the advantage of capturing complex user preferences and working well with sparse data. However, it faces challenges such as the cold start problem for new users or items and scalability issues with large datasets.

In our movie recommendation system, let's consider a user, Alice, who has rated several movies highly, including *The Shawshank Redemption* (5 stars), *Pulp Fiction* (4 stars), and *The Godfather* (5 stars). User-based CF would find users with similar rating patterns to Alice, such as Bob, who has rated these same movies similarly and gave a high rating to *Inception*. So, the system might recommend *Inception* to Alice based on Bob's high rating. Item-based CF, on the other hand, would find movies similar to those Alice rated highly. It might determine that users who liked *The Shawshank Redemption* also enjoyed *The Green Mile*, leading to a recommendation for Alice.

- **Content-based filtering** recommends items based on their features and the user's past preferences. It builds a profile for each user and item based on their characteristics and uses similarity measures between user profiles and item features to generate recommendations. This approach has the advantage of providing explanations for recommendations and working well for niche items. However, it has limitations in expanding a user's interests and requires rich item metadata. In our movie example, we'd analyze the features of movies Alice likes, such as *The Shawshank Redemption* (drama, directed by Frank Darabont, starring Tim Robbins and Morgan Freeman, released in 1994) and *The Godfather* (crime/drama, directed by Francis Ford Coppola, starring Marlon Brando and Al Pacino, released in 1972). In this case, the system might recommend other drama movies from the 1970s or 1990s, or films featuring Morgan Freeman or Al Pacino.
- **Hybrid systems** combine multiple recommendation approaches to leverage their strengths and mitigate weaknesses. Common hybrid strategies include weighted approaches that combine scores from different recommenders, switching methods that choose between different recommenders based on the context, and feature combination techniques that use features from different sources as input to a single recommender. Hybrid systems often provide better performance but come with increased complexity in terms of their design and implementation. In our example, a hybrid approach might combine CF and content-based filtering. It could use CF to find similar users to Alice and then filter those recommendations based on content features she prefers, such as favoring dramas or movies from certain decades.

Table 9.1 provides a comparison of the different types, highlighting their key characteristics, advantages, and limitations:

Feature	**CF**	**Content-based filtering**	**Hybrid systems**
Data required	User-item interactions	Item features and user preferences	Both types of data
Strengths	- Captures user preferences well - Discovers new interests - Works with minimal item information	- No cold start problem for items - Can explain recommendations - Good for niche items	- Better accuracy - Overcomes limitations of individual methods - More robust

Feature	CF	Content-based filtering	Hybrid systems
Weaknesses	- Cold start problem - Sparsity issues - Limited for new items	- Limited serendipity - Requires detailed item features - Over-specialization	- Complex implementation - Computationally expensive - Requires more data
Best use cases	E-commerce, social media	News articles, specialized content	Large-scale platforms, streaming services

Table 9.1 – Comparing different recommendation system types

Key metrics and evaluation

Evaluating the performance of recommendation systems is crucial for understanding their effectiveness and guiding improvements. Several metrics and evaluation techniques are commonly used.

Accuracy metrics

This is how we define these commonly used metrics:

- **Precision** measures the proportion of relevant items among the recommended items:
  - Precision = (Number of relevant recommendations) / (Total number of recommendations)
- **Recall** measures the proportion of relevant items that were successfully recommended:
  - Recall = (Number of relevant recommendations) / (Total number of relevant items)
- **F1 score** is the harmonic mean of precision and recall, providing a balanced measure:
  - F1 = 2 × (Precision × Recall) / (Precision + Recall)
- The **mean average precision** (**MAP**) calculates the mean of the average precision scores for each user:
  - $MAP = \left(\frac{1}{|U|} \cdot \Sigma(AP(u))\right)$, where we have the following:
    - $|U|$ is the number of users.
    - $AP(u)$ is the average precision of the user, u.
    - $AP = \Sigma(P(k) \times rel(k))$ / number of relevant items
    - $P(k)$ is the precision at the cutoff, k.
    - $rel(k)$ is the indicator function (1 if item k is relevant, 0 otherwise).

- The **normalized discounted cumulative gain** (**NDCG**) measures the quality of ranking while taking into account the position of relevant items in the recommendation list:
    - $NDCG = DCG / IDCG$, where we have the following:
        - $DCG(Discounted\ Cumulative\ Gain) = \Sigma\left(rel_i/\log_2(i+1)\right)$
        - $IDCG$ is the ideal DCG (max possible DCG).
        - rel_i is the relevance score of the item at position i.
        - i is the position in the ranked list.

Assuming we recommend five movies to Alice, and she ends up liking three of them, the precision would be 3/5 = 0.6 or 60%. If there were 10 movies in total that Alice would have liked, the recall would be 3/10 = 0.3 or 30%. The F1 score, which provides a balanced measure, would be approximately 0.4 or 40% in this case.

Error metrics

Error metrics often measure the average difference between predicted and actual ratings, such as the following:

- The **mean absolute error** (**MAE**) measures the average absolute difference between predicted and actual ratings:
    - $MAE = \Sigma|\text{actual} - \text{predicted}|/n$, where n is the number of predictions
- The **root mean squared error** (**RMSE**) is similar to MAE but penalizes large errors more heavily:
    - $RMSE = \sqrt{}(\Sigma(\text{actual} - \text{predicted})^2/n)$

For example, if we predicted ratings for five movies and the average difference between our predictions and Alice's actual ratings was 0.42 stars, this would be our MAE. This suggests that, on average, our predictions are off by about 0.42 stars.

Coverage, diversity, and serendipity

Coverage metrics assess the system's ability to recommend a wide range of items. **Item coverage** measures the percentage of items in the catalog that the system can recommend, while **user coverage** indicates the percentage of users for whom the system can generate recommendations. In our movie system, we might measure what percentage of the 3,000 movies in our catalog we're able to recommend (item coverage) and what percentage of our 1,000 users receive recommendations (user coverage).

Meanwhile, diversity metrics evaluate the variety (or dissimilarity) among recommended items, ensuring users receive a range of suggestions rather than similar ones. **Serendipity**, on the other hand, assesses the system's ability to provide unexpected yet relevant recommendations, creating pleasant surprises for users. It combines elements of unexpectedness and relevance, often measured through unexpectedness scores and relevance evaluations. While related, diversity and serendipity are distinct: diverse recommendations may not always be serendipitous, and serendipitous recommendations often contribute to diversity but not vice versa.

These metrics are essential for creating a balanced recommendation system that offers varied, interesting, and delightfully unexpected suggestions to users (for more on this topic, see `https://doi.org/10.1145/2926720`).

A/B testing and user feedback

While offline metrics provide valuable insights, real-world performance is best evaluated through **A/B testing** and **user feedback**. A/B testing involves comparing two versions of the recommendation system with real users and measuring key performance indicators such as click-through rate, conversion rate, or user engagement. User feedback, both explicit (through ratings or surveys) and implicit (through clicks or purchase behavior), is crucial for continuously improving and validating the recommendation system. Long-term user satisfaction, which is measured through user retention and engagement metrics, is also essential for assessing the impact of recommendations on user trust and platform loyalty.

Evaluating recommendation systems requires a holistic approach that considers both offline metrics and real-world performance. As recommendation systems become more sophisticated, especially with the integration of graph deep learning techniques, evaluation methods continue to evolve to capture the nuanced aspects of recommendation quality and user satisfaction.

Graph structures in recommendation systems

Graph structures have emerged as a powerful paradigm for modeling complex relationships in recommendation systems. By representing users, items, and their interactions as nodes and edges in a graph, we can capture rich, multi-dimensional information that traditional matrix-based approaches often miss.

User-item interaction graphs

The foundation of graph-based recommendation systems is the **user-item interaction graph**. In this structure, users and items are represented as *nodes*, while interactions (such as ratings, views, or purchases) form *edges* between them.

For a movie recommendation system, the graph might look like this:

- **Nodes**: Users (U1, U2, U3…) and movies (M1, M2, M3…)
- **Edges**: Ratings or views (for example, U1 -> M1 with a weight of 4 stars)

This simple structure already allows for a more nuanced analysis than a traditional user-item matrix. For example, we can easily identify the following:

- Popular movies (nodes with many incoming edges)
- Active users (nodes with many outgoing edges)
- Similar users or movies (nodes with similar edge patterns)

Consider user U1 who has watched and rated movies M1 (4 stars), M2 (3 stars), and M3 (5 stars). To recommend a new movie to U1, we can traverse the graph to find users with similar rating patterns and suggest movies they've enjoyed that U1 hasn't seen yet.

Incorporating side information

Real-world recommendation systems often have access to additional information beyond just user-item interactions. Graph structures excel at incorporating this side information seamlessly. For movie recommendations, we might enhance our graph with the following aspects:

- **Movie attributes**:
    - Genres (action, comedy, drama, and so on)
    - Directors
    - Actors
    - Release year
- **User attributes**:
    - Age
    - Gender
    - Location
- **Social connections**:
    - Friend relationships between users

These additional nodes and edges create a heterogeneous graph, where different types of nodes and relationships coexist. This rich structure allows for more sophisticated recommendation algorithms. Let's expand on our previous example. Now, movie M1 has additional connections:

- **Genre**: Action
- **Director**: D1
- **Actors**: A1, A2

User U1 is connected to the following aspects:

- **Age group**: 25-34
- **Location**: New York

By traversing this enhanced graph, we can make more nuanced recommendations. For instance, we might suggest an action movie directed by D1 that's popular among U1's age group in New York, even if it doesn't have a direct connection to U1's previously watched movies.

Temporal graphs

Time is a crucial factor in recommendation systems, especially for domains such as movies, where preferences can change rapidly. **Temporal graphs** incorporate time information into the graph structure.

There are several ways to represent time in a graph:

- **Time-stamped edges**: Each interaction edge includes a timestamp
- **Time-based node splitting**: Create multiple nodes for the same user or item at different time points
- **Dynamic graphs**: The graph structure itself evolves, with nodes and edges appearing or disappearing

For movie recommendations, a temporal graph can capture the following:

- Changes in user preferences over time
- The life cycle of movie popularity
- Seasonal trends in viewing habits

Consider a user, U1, who watched mostly comedies in 2020 but shifted toward dramas in 2021. A temporal graph would preserve this information, allowing the recommendation system to prioritize recent preferences while still considering long-term patterns.

We might represent this as follows:

```
U1_2020 -> M1 (Comedy, watched Jan 2020)
U1_2020 -> M2 (Comedy, watched Jun 2020)
U1_2021 -> M3 (Drama, watched Jan 2021)
U1_2021 -> M4 (Drama, watched May 2021)
```

This structure enables sequential recommendation, where the system suggests the next movie based on the user's recent viewing history and overall trends.

Multi-relational graphs

In complex domains such as movie recommendations, different types of relationships often coexist. **Multi-relational graphs** allow us to represent these varied connections explicitly.

For movies, we might have these relationships:

- **User-movie**: `"rated"`, `"watched"`, and `"added to wishlist"`
- **User-user**: `"is friends with"` and `"has similar taste to"`
- **Movie-movie**: `"has same genre"`, `"has same director"`, and `"is sequel to"`

Each relationship can have its own semantics and importance in the recommendation process. Here's an example:

```
U1 --("rated", 5)-> M1
U1 --("is friends with")-> U2
U2 --("rated", 4)-> M2
M1 --("has same director")-> M2
```

This multi-relational structure allows for sophisticated path-based recommendations. For instance, we might recommend M2 to U1 because of the following reasons:

- U1's friend, U2, rated it highly.
- It shares a director with M1, which U1 loved.

By leveraging these complex relationships, multi-relational graphs enable more contextual and explainable recommendations.

Graph structures excel at capturing complex relationships, incorporating diverse information sources, and modeling temporal dynamics. As we'll see in subsequent sections, these rich graph structures form the foundation for sophisticated deep learning models that can generate highly personalized and accurate recommendations.

Graph-based recommendation models

Graph-based recommendation models have emerged as powerful tools for capturing complex relationships between users and items in recommendation systems. These models leverage the rich structural information inherent in user-item interaction graphs to generate more accurate and personalized recommendations. In this section, we'll explore three major categories of graph-based recommendation models, starting with **matrix factorization** (**MF**) with graph regularization.

MF with graph regularization

MF is a fundamental technique in CF, and its integration with graph structures has led to significant improvements in recommendation quality. **Graph regularization** in MF models helps to incorporate the structural information of the user-item interaction graph into the learning process.

The basic idea is to add a regularization term to the traditional MF objective function that encourages connected nodes in the graph to have similar latent representations. This approach helps to capture the local structure of the user-item graph and can lead to more accurate recommendations.

For movie recommendations, consider a scenario where we have a user-item graph with users and movies as nodes, and ratings as edges. The graph regularization term would encourage users who have rated similar movies to have similar latent factors, and movies that have been rated similarly to also have similar latent factors.

A typical objective function for MF with graph regularization might look like this:

$$L = \Sigma_{(u,i) \in o} \left(r_{ui} - p_u^T q_i\right)^2 + \lambda\left(\|P\|_F^2 + \|Q\|_F^2\right) + \alpha \Sigma_{(u,v) \in \mathcal{E}_u} \| p_u - p_v \|^2 + \beta \Sigma_{(i,j) \in \mathcal{E}_i} \|q_i - q_j\|^2$$

Here, p_u and q_i are the latent factors for the user, u, and the item, i, respectively, $\mathcal{E}_u$ and $\mathcal{E}_i$ are the sets of edges in the user-user and item-item graphs, and α and β are regularization parameters.

This approach has shown improved performance over traditional MF, especially in scenarios with sparse data, as it can leverage the graph structure to make better predictions for users or items with few interactions.

Graph neural network models

As we saw in *Chapter 4*, **graph neural networks** (**GNNs**) have revolutionized the field of graph-based recommendations by enabling direct learning on graph-structured data. The following models can capture high-order connectivity patterns and learn rich node representations that incorporate both node features and graph structure.

PinSage

PinSage, developed by Pinterest, is a pioneering GNN model for large-scale recommendation systems. It adapts the **GraphSAGE** architecture to generate embeddings efficiently for nodes in web-scale graphs.

In a movie recommendation context, PinSage could be used to generate embeddings for both users and movies. The model would aggregate information from a node's local neighborhood, capturing not just direct interactions but also higher-order relationships. For example, it could identify that two users who haven't watched the same movies might still have similar tastes if they've watched movies that are themselves similar.

The key steps in PinSage are as follows:

1. **Neighborhood sampling**: For each node, sample a fixed number of neighbors to make computation feasible on large graphs.
2. **Feature aggregation**: Aggregate features from the sampled neighbors using learnable aggregation functions.
3. **Embedding generation**: Combine the aggregated neighborhood information with the node's features to generate the final embedding.

Neural graph collaborative filtering

Neural graph collaborative filtering (**NGCF**) explicitly incorporates the user-item graph structure into the embedding learning process. It propagates embeddings on the user-item interaction graph to capture collaborative signals.

For movie recommendations, NGCF would start with initial embeddings for users and movies and then update these embeddings iteratively by passing messages along the edges of the user-movie interaction graph. This process allows the model to capture high-order connectivity between users and movies.

The embedding update process in NGCF can be described as follows:

$$e_u^{(k)} = \sigma\left(W_1^{(k)} e_u^{(k-1)} + \sum_{i \in N_u} \frac{1}{\sqrt{|N_u||N_i|}} W_2^{(k)} e_i^{(k-1)}\right)$$

Here, $e_u^{(k)}$ is the embedding of the user, u, at the kth layer. N_u is the set of items that are interacted with by the user, u, and $W_1^{(K)}$and $W_2^{(K)}$ are learnable weight matrixes.

LightGCN

LightGCN simplifies the NGCF model by removing feature transformation and nonlinear activation, focusing solely on the most essential component of **graph convolutional networks** (**GCNs**) for CF: neighborhood aggregation.

In the context of movie recommendations, LightGCN would represent users and movies as nodes in a bipartite graph. The model then performs multiple layers of neighborhood aggregation to capture high-order connectivity. The final embeddings are a weighted sum of embeddings from all layers.

The embedding propagation rule in LightGCN is as follows:

$$e_u^{(k=1)} = \sum_{i \in N_u} \frac{1}{\sqrt{|N_u||N_i|}} e_i^{(k)}$$

The simplicity of LightGCN makes it computationally efficient while still achieving state-of-the-art performance on many recommendation tasks.

Training graph deep learning models

Training graph deep learning models for recommendation systems is a complex process that requires various factors to be considered carefully. In this section, we'll provide a comprehensive guide to training these models.

Data preprocessing

Effective data preprocessing is crucial for the success of graph-based recommendation models. Let's dive deeper into the steps involved.

Building the interaction graph

Creating a high-quality interaction graph is the foundation of graph-based recommendation systems. Let's take a look:

1. **Node creation**:

- Assign unique identifiers to each user and movie.
- Create node attributes to store relevant information.

2. **Edge creation**:

- Create edges based on user-movie interactions.
- Consider different types of interactions (for example, ratings, views, and likes).

3. **Edge weighting**:

- Assign weights to edges based on interaction strength.
- Normalize weights to ensure consistency across different interaction types.

4. **Handling temporal information**:

- Incorporate the timestamps of interactions as edge attributes.
- Consider creating multiple edges for repeated interactions.

Here's some example pseudocode for building a more detailed graph:

```
import networkx as nx
from datetime import datetime

def build_movie_graph(interactions):
    G = nx.Graph()
    for user_id, movie_id, rating, timestamp in interactions:
        # Add user node with type attribute
        user_node = f"user_{user_id}"
        G.add_node(user_node, type="user",
                   last_active=timestamp)

        # Add movie node with type attribute
        movie_node = f"movie_{movie_id}"
        G.add_node(movie_node, type="movie")

        # Add or update edge
        if G.has_edge(user_node, movie_node):
            G[user_node][movie_node]['weight'] += 1
            G[user_node][movie_node]['last_interaction'] = timestamp
            G[user_node][movie_node]['ratings'].append(float(rating))
        else:
            G.add_edge(user_node, movie_node,
                       weight=1,
                       last_interaction=timestamp,
                       ratings=[float(rating)])
    return G
```

The following function normalizes the edge weights in the graph to provide a relative measure of interaction strength:

```
def normalize_edge_weights(G):
    for edge in G.edges():
        ratings = G.edges[edge].get('ratings', [])
        if ratings:
            avg_rating = np.mean(ratings)
            # Store average rating:
            G.edges[edge]['rating'] = avg_rating
            # Normalize to [0,1]:
            G.edges[edge]['weight'] = avg_rating / 5.0
        else:
            G.edges[edge]['rating'] = 0
            G.edges[edge]['weight'] = 0
```

Feature engineering

You can enhance the graph with rich features to improve model performance:

- **User features**:
    - Demographic information, such as age, gender, and location
    - Behavioral features, such as average rating, genre preferences, and activity level
    - Derived features, such as user segments based on viewing patterns
- **Movie features**:
    - Basic attributes, such as genre, release year, and duration
    - Production details, such as director, actors, budget, and production company
    - Performance metrics, such as box office revenue and critic ratings
    - Derived features, such as popularity score and genre embeddings
- **Temporal features**:
    - Time-based user features, such as viewing frequency and time since the last activity
    - Time-based movie features, such as the age of the movie and seasonal popularity
- **Graph-based features**:
    - Node degree, such as the number of movies rated by a user or the number of ratings for a movie
    - Centrality measures, such as PageRank and betweenness centrality
    - Community detection – for example, assign community labels to nodes

Here's an example of advanced feature engineering:

```
import numpy as np
from sklearn.preprocessing import StandardScaler

def engineer_user_features(G, user_data):
    user_features = {}
    for node, data in G.nodes(data=True):
        if data['type'] == 'user':
            user_id = node.split('_')[1]
            # Add check if user exists in user_data:
            if user_id in user_data:
                user_info = user_data[user_id]
                # Basic features
                features = [
                    float(user_info['age']),  # Convert to float
                    float(user_info['gender_encoded']),
                    float(user_info['location_encoded']),
                ]

                # Behavioral features
                # Get ratings directly from edges
                ratings = [G[node][edge].get(
                    'rating', 0) for edge in G[node]]
                avg_rating = np.mean(ratings) if ratings else 0
                rating_count = G.degree(node)
                features.extend([avg_rating, float(rating_count)])

                # Add genre preferences
                genre_preferences = calculate_genre_preferences(
                    G, node)
                features.extend(genre_preferences)

                user_features[node] = np.array(
                    features, dtype=np.float32)

    return user_features
```

The following function calculates genre preferences for a given user. This is used as part of the feature engineering process outlined previously:

```
def calculate_genre_preferences(G, user_node):
    genre_counts = {genre: 0 for genre in GENRE_LIST}
    total_ratings = 0

    # Iterate through neighboring movie nodes
    for neighbor in G[user_node]:
        if neighbor.startswith('movie_'):
            movie_id = neighbor.split('_')[1]
            if movie_id in movie_data:
                genres = movie_data[movie_id]['genres']
                rating = G[user_node][neighbor]['rating']
                for genre in genres:
                    if genre in genre_counts:
                        genre_counts[genre] += rating
                total_ratings += 1

    # Normalize genre preferences
    genre_preferences = []
    for genre in GENRE_LIST:
        if total_ratings > 0:
            genre_preferences.append(
                genre_counts[genre] / total_ratings)
        else:
            genre_preferences.append(0)

    return genre_preferences
```

Model training techniques

Training graph deep learning models for recommendation systems involves several advanced techniques to improve performance and efficiency.

Loss functions

You can choose and combine appropriate **loss functions** based on the recommendation task:

- **Binary cross-entropy** (**BCE**) can be used for *implicit* feedback:

```
def bce_loss(predictions, targets):
    return F.binary_cross_entropy_with_logits(
        predictions, targets)
```

- **Mean squared error** (**MSE**) can be employed for *explicit* feedback (rating prediction):

```
def mse_loss(predictions, targets):
    return F.mse_loss(predictions, targets)
```

- **Bayesian personalized ranking** (**BPR**) **loss** can be utilized for pairwise ranking:

```
def bpr_loss(pos_scores, neg_scores):
    return -F.logsigmoid(pos_scores - neg_scores).mean()
```

- **Margin ranking loss** is another option for pairwise ranking:

```
def margin_ranking_loss(pos_scores, neg_scores, margin=0.5):
    return F.margin_ranking_loss(
        pos_scores, neg_scores,
        torch.ones_like(pos_scores), margin=margin)
```

- **Combination of losses** can combine multiple loss functions for multi-task learning:

```
def combined_loss(pred_ratings, true_ratings,
                  pos_scores, neg_scores, alpha=0.5):
# Convert inputs to floating point tensors if they aren't
```

```
already
    pred_ratings = pred_ratings.float()
    true_ratings = true_ratings.float()
    pos_scores = pos_scores.float()
    neg_scores = neg_scores.float()

    rating_loss = mse_loss(pred_ratings, true_ratings)
    ranking_loss = bpr_loss(pos_scores, neg_scores)
    return alpha * rating_loss + (1 - alpha) * ranking_loss
```

Training loop

You can also implement an advanced **training loop** with various optimization techniques:

- **Gradient accumulation for larger effective batch sizes**: This technique allows for larger effective batch sizes by accumulating gradients over multiple batches before performing an optimizer step:

```
# Inside the training loop
for batch_idx, batch in enumerate(generate_batches(train_
graph)):
    # ... (model forward pass and loss calculation)

    # Gradient accumulation
    loss = loss / ACCUMULATION_STEPS
    loss.backward()

    if (batch_idx + 1) % ACCUMULATION_STEPS == 0:
        optimizer.step()
        optimizer.zero_grad()
```

- **Learning rate scheduling**: The `ReduceLROnPlateau` scheduler adjusts the learning rate based on the validation loss, reducing it when the loss plateaus:

```
scheduler = ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=3
)
# Inside the training loop
scheduler.step(val_loss)
```

- **Early stopping based on validation performance**: This technique stops training when the validation loss doesn't improve for a specified number of epochs, preventing overfitting:

```
# Inside the training loop
if val_loss < best_val_loss:
    best_val_loss = val_loss
    no_improve_count = 0
```

```
        torch.save(model.state_dict(), 'best_model.pth')
    else:
        no_improve_count += 1
        if no_improve_count >= patience:
            print(f"Early stopping after {epoch} epochs")
            break
    # After training
    model.load_state_dict(torch.load('best_model.pth'))
```

- **Gradient clipping to prevent exploding gradients**: Gradient clipping prevents gradients from becoming too large, which can cause instability during training:

```
# Inside the training loop, before optimizer step
torch.nn.utils.clip_grad_norm_(
    model.parameters(), max_norm=1.0)
```

Scalability and optimization

Handling large-scale graphs requires advanced techniques for efficient training and inference. We covered the challenge of scalability in *Chapter 5*; here, we'll look at practical examples of techniques that can help us address this issue.

Mini-batch training with neighborhood sampling

Instead of processing the entire graph, we can use **mini-batch training** with neighborhood sampling:

1. Sample a subset of user nodes:

```
def create_mini_batch(G, batch_size, n_pos=5, n_neg=5,
                      n_neighbors=10, n_hops=2):
    # Get all user nodes
    all_user_nodes = [n for n in G.nodes() if
                      n.startswith('user_')]
    if not all_user_nodes:
        raise ValueError("No user nodes found in graph")

    # Randomly select users
    user_nodes = random.sample(
        all_user_nodes,
        min(batch_size, len(all_user_nodes))
    )

    # Create subgraph
    subgraph = G.subgraph(user_nodes).copy()
```

This step randomly selects a subset of user nodes to form the mini-batch.

2. For each user, sample a fixed number of positive and negative movie interactions. Here, we sample positive movie interactions from the user's neighbors and negative interactions from movies the user hasn't interacted with:

```
    # Sample positive and negative movies
    pos_movies = []
    neg_movies = []
    all_movies = [n for n in G.nodes() if
                  n.startswith('movie_')]

    for user in user_nodes:
        user_movies = [n for n in G[user] if
                       n.startswith('movie_')]
        pos_sample = random.sample(
            user_movies, min(n_pos, len(user_movies))
        ) if user_movies else []
        available_neg = list(set(all_movies) -
                             set(user_movies))
        neg_sample = random.sample(available_neg, min(
            n_neg, len(available_neg))
        ) if available_neg else []

        pos_movies.extend(pos_sample)
        neg_movies.extend(neg_sample)

    return subgraph, user_nodes, pos_movies, neg_movies
```

3. Perform **multi-hop neighborhood sampling** to create a subgraph containing the relevant nodes and their neighbors:

```
def sample_neighbors(graph, nodes, n_neighbors, n_hops):
    sampled_nodes = set(nodes)
    for _ in range(n_hops):
        new_nodes = set()
        for node in sampled_nodes:
            neighbors = list(graph.neighbors(node))
            sampled = random.sample(
                neighbors, min(n_neighbors, len(neighbors))
            )
            new_nodes.update(sampled)
        sampled_nodes.update(new_nodes)
    return list(sampled_nodes)
    # Inside create_mini_batch function
    all_nodes = user_nodes + pos_movies + neg_movies
    sampled_nodes = sample_neighbors(
        graph, all_nodes, n_neighbors, n_hops)
```

4. Finally, we create the subgraph and conduct message passing within the sampled subgraph:

```
subgraph = graph.subgraph(sampled_nodes)
return subgraph, user_nodes, pos_movies, neg_movies
```

Distributed training

For extremely large graphs, you can implement **distributed training**:

1. **Graph partitioning**: Divide the graph across multiple machines to enable distributed processing:

```
def partition_graph(graph, rank, world_size):
    # Implement graph partitioning logic here
    # This is a placeholder function
    num_nodes = graph.number_of_nodes()
    nodes_per_partition = num_nodes // world_size
    start_node = rank * nodes_per_partition
    end_node = start_node + nodes_per_partition if rank < \
        world_size - 1 else num_nodes
    local_nodes = list(graph.nodes())[start_node:end_node]
    return graph.subgraph(local_nodes)
# Usage in distributed_train function
local_graph = partition_graph(graph, rank, world_size)
```

2. **Distributed message passing**: Implement efficient communication protocols for distributed message passing across machines:

```
import torch.distributed as dist

def distributed_message_passing(local_graph, node_features):
    # Perform local message passing
    local_output = local_message_passing(
        local_graph, node_features)

    # Gather results from all processes
    gathered_outputs = [torch.zeros_like(local_output) for _ in
                        range(dist.get_world_size())]
    dist.all_gather(gathered_outputs, local_output)

    # Combine gathered results
    global_output = torch.cat(gathered_outputs, dim=0)
```

```
        return global_output

    # Usage in model forward pass
    def forward(self, graph, node_features):
        # ... other layers
        node_features = distributed_message_passing(
            graph, node_features)
        # ... remaining layers
```

3. **Parameter server architecture**: Use **DistributedDataParallel** (**DDP**) to centralize model parameter updates across all processes:

```
from torch.nn.parallel import DistributedDataParallel as DDP

def distributed_train(rank, world_size, graph):
    setup(rank, world_size)

    # Create model and move it to GPU with id rank
    model = GraphRecommender(...).to(rank)
    model = DDP(model, device_ids=[rank])

    optimizer = optim.Adam(model.parameters(), lr=0.001)

    # ... training loop

    cleanup()
```

By implementing all the strategies outlined in this section, such as efficient graph construction, comprehensive feature engineering, sophisticated loss functions, and scalable training methods, you can develop powerful and effective graph-based recommendation systems.

Explainability in graph-based recommendations

As recommendation systems become more sophisticated, the need for **explainable AI** (**XAI**) in these systems grows. Graph-based models offer unique opportunities for enhancing the explainability of recommendations.

Attention mechanisms for interpretability

As we saw in *Chapter 4*, **graph attention networks** (**GATs**) can be leveraged to provide insights into which nodes or features contribute most to a recommendation. For movie recommendations, this could reveal which actors, directors, or genres have the most significant influence on a user's preferences.

Consider a user who frequently watches action movies starring Tom Cruise. The attention mechanism might highlight that the presence of Tom Cruise in a movie's cast graph node has a higher weight in the recommendation process for this user compared to other factors.

Path-based explanations

Metapath-based models can offer intuitive explanations by showing the reasoning path that led to a recommendation.

For example, a movie recommendation might be explained as follows: *We recommended "Inception" because you enjoyed "The Dark Knight" (same director: Christopher Nolan) and "Interstellar" (similar genre: sci-fi thriller).*

One of the major challenges that businesses face is a lack of data. A new product might not have enough customer data to build these sophisticated recommendation models. This challenge is called the cold start problem, and we're going to look into how we can leverage graph algorithms to solve it.

The cold start problem

The cold start problem in recommendation systems refers to the challenge of making accurate recommendations for new users or new items that have little to no interaction data. In movie recommendation systems, this occurs when a new user joins the platform and has no viewing history or when a new movie is released and has no user ratings or interactions.

The cold start problem is particularly challenging in movie recommendation systems, especially for new users or newly released movies.

Graph embedding transfer

One solution to the cold start problem is **graph embedding transfer**, a technique that's used to initialize representations for new nodes (movies or users) in a recommendation graph when there's no interaction data available. Here's a general description:

- For new items (for example, movies):
    - Identify similar existing items based on metadata or content features.
    - Use the embeddings of these similar items to initialize the embedding of the new item.
    - This gives the new item a starting point in the embedding space that reflects its likely characteristics.

- For new users:
  - Use available demographic or preference information to find similar existing users.
  - Initialize the new user's embedding based on these similar users' embeddings.
  - As the new user interacts with the system, their embedding can be fine-tuned.

For example, when a new superhero movie is released, we can initialize its embedding by averaging the embeddings of other superhero movies in the graph:

```
def initialize_new_movie(movie_id, similar_movies, movie_embeddings):
    if not similar_movies:
        return np.zeros(next(iter(movie_embeddings.values())).shape)

    similar_embeddings = [
        movie_embeddings[m] for m in similar_movies if
        m in movie_embeddings
    ]
    if not similar_embeddings:
        return np.zeros(next(iter(movie_embeddings.values())).shape)

    return np.mean(similar_embeddings, axis=0)
```

Content-based feature integration

Another effective approach to mitigate cold start issues is to leverage content-based features from available metadata or item descriptions. By integrating multiple content-based features such as text attributes, categorical information, and any available numerical data, a system can generate initial recommendations even for new items or users with no interaction history. This integrated feature representation can be used to compute item similarities or train machine learning models that predict user preferences based on item characteristics.

For instance, when a new user signs up, we can create edges to movies based on their stated preferences (genres, actors, and directors), even before they've watched anything:

```
def create_initial_edges(user_id, preferences, movie_graph):
    for pref in preferences:
        similar_movies = [
            m for m in movie_graph.nodes if
            pref in movie_graph.nodes[m]['attributes']]
        for movie in similar_movies:
            movie_graph.add_edge(
                user_id, movie, weight=0.5 # Initial weak connection
            )
    return movie_graph

    # Usage
    updated_graph = create_initial_edges(
        'new_user_123', ['Sci-Fi', 'Tom Hanks'], movie_graph)
```

These are just a couple of ways we can approach a cold start problem in a recommendation system that leverages graphs.

Summary

In this chapter, we introduced graph deep learning as an advanced approach to recommendation systems. You learned about the fundamental concepts of recommendation systems, including the different types and evaluation metrics.

Then, we delved into graph structures for representing user-item interactions, incorporating side information, and capturing temporal dynamics. Various graph-based recommendation models were explored, from MF with graph regularization to advanced GNN models. You also became familiar with a variety of training techniques, scalability challenges, and advanced topics such as explainability and the cold start problem in graph-based recommendation systems.

In the next chapter, we're going to investigate the applications of graph learning in computer vision.

10
Graph Deep Learning for Computer Vision

Computer vision (**CV**) has traditionally relied on grid-based representations of images and videos, which have been highly successful with **convolutional neural networks** (**CNNs**). However, many visual scenes and objects have inherent relational and structural properties that aren't easily captured by grid-based approaches. This is where graph representations come into play, offering a more flexible and expressive way to model visual data.

Graphs can naturally represent relationships between objects in a scene, hierarchical structures in images, non-grid data such as 3D point clouds, and long-range dependencies in videos. For example, in a street scene, a graph can represent cars, pedestrians, and traffic lights as nodes, with edges representing their spatial relationships or interactions. This representation captures the scene's structure more intuitively than a pixel grid.

In this chapter, we'll elaborate on the following topics:

- Traditional CV approaches versus graph-based approaches
- Graph construction for visual data
- **Graph neural networks** (**GNNs**) for image classification
- Object detection and segmentation using GNNs
- Multi-modal learning with GNNs
- Limitations and next steps

Traditional CV approaches versus graph-based approaches

Traditional CV approaches primarily rely on CNNs that operate on regular grid structures, extracting features through convolution and pooling operations. While effective for many tasks, these methods often struggle with long-range dependencies and relational reasoning. In contrast, graph-based approaches represent visual data as nodes and edges, utilizing GNNs to process information. This structure allows for easier incorporation of non-local information and relational inductive biases.

For instance, in image classification, a CNN might have difficulty relating distant parts of an image, whereas a graph-based approach could represent different image regions as *nodes* and their relationships as *edges*, facilitating long-range reasoning. This fundamental difference in data representation and processing enables graph-based methods to overcome some of the limitations inherent in traditional CNN-based approaches, potentially leading to improved performance in tasks that require understanding complex spatial relationships or global context within visual data.

The advantages of graph representations for visual data are as follows:

- **Flexibility**: Graphs can represent various types of visual data, from pixels to objects to entire scenes.
- **Relational reasoning**: Graphs explicitly model relationships, making it easier to reason about object interactions.
- **Incorporating prior knowledge**: Domain knowledge can be easily encoded in the graph structure.
- **Handling irregular data**: Graphs are well suited for non-grid data such as 3D point clouds or social network images.
- **Interpretability**: Graph structures often align more closely with human understanding of visual scenes.

For instance, in a face recognition task, a graph-based approach might represent facial landmarks (specific points on a face that correspond to key facial features such as the corners of the eyes, the tip of the nose, the edges of the mouth, and so on) as *nodes* and their geometric relationships as *edges*. This representation can be more robust to variations in pose and expression compared to grid-based approaches.

Here's a simple example of constructing a face graph:

```
import networkx as nx

def create_face_graph(landmarks, threshold=2.0):
    G = nx.Graph()
    for i, landmark in enumerate(landmarks):
        G.add_node(i, pos=landmark)

    # Connect nearby landmarks
    for i in range(len(landmarks)):
        for j in range(i+1, len(landmarks)):
            if np.linalg.norm(landmarks[i] - landmarks[j]) < \
                    threshold:
                G.add_edge(i, j)
    return G

# Usage
landmarks = np.array([[x1, y1], [x2, y2], ...])  # facial landmark
coordinates
face_graph = create_face_graph(landmarks)
```

This graph representation captures the spatial relationships between facial features, which can be leveraged by GNNs for tasks such as face recognition or emotion detection. Now, let's jump into the concepts of constructing graphs specifically for image data.

Graph construction for visual data

Constructing graphs from visual data is a crucial step in applying graph-based methods to CV tasks. The choice of graph construction method can significantly impact the performance and interpretability of downstream tasks. This section explores various approaches to graph construction, each suited to different types of visual data and problem domains.

Pixel-level graphs

Pixel-level graphs represent images at their most granular level, with each pixel serving as a *node* in the graph. *Edges* are typically formed between neighboring pixels, creating a grid-like structure that mirrors the original image. This approach preserves fine-grained spatial information but can lead to large, computationally expensive graphs for high-resolution images.

For example, in a 100x100 pixel image, we would create a graph with 10,000 nodes. Each node might be connected to its four or eight nearest neighbors, depending on whether we consider diagonal connections. The node features could include color information (RGB values) and pixel coordinates. This type of graph is particularly useful for tasks that require precise spatial information, such as image segmentation or edge detection.

Here's a simple example of how you might construct a pixel-level graph using **NetworkX**:

```
import networkx as nx
import numpy as np

def create_pixel_graph(image, connectivity=4):
    height, width = image.shape[:2]
    G = nx.Graph()
    for i in range(height):
        for j in range(width):
            node_id = i * width + j
            G.add_node(node_id, features=image[i, j], pos=(i, j))

            if connectivity == 4:
                neighbors = [(i-1, j), (i+1, j),
                             (i, j-1), (i, j+1)]
            elif connectivity == 8:
                neighbors = [(i-1, j), (i+1, j), (i, j-1),
                             (i, j+1), (i-1, j-1), (i-1, j+1),
                             (i+1, j-1), (i+1, j+1)]

            for ni, nj in neighbors:
                if 0 <= ni < height and 0 <= nj < width:
                    neighbor_id = ni * width + nj
                    G.add_edge(node_id, neighbor_id)

    return G
```

This function creates a graph representation of an image, where each pixel is a node and edges connect neighboring pixels based on the specified connectivity (`4` or `8`).

Let's call the function:

```
image = np.random.rand(100, 100, 3)  # Random RGB image
pixel_graph = create_pixel_graph(image, connectivity=8)
```

Superpixel-based graphs

Superpixel-based graphs offer a middle ground between pixel-level and object-level representations. Superpixels are groups of pixels that share similar characteristics, often created through image segmentation algorithms such as **simple linear iterative clustering** (**SLIC**). In a superpixel graph, each *node* represents a superpixel, and *edges* connect adjacent superpixels.

This approach reduces the graph size compared to pixel-level graphs while still maintaining local image structure. For instance, a 1,000x1,000-pixel image might be reduced to a graph of 1,000 superpixels, each representing an average of 1,000 pixels. Node features could include average color, texture information, and the spatial location of the superpixel.

Superpixel graphs are particularly effective for tasks such as semantic segmentation or object proposal generation. They capture local consistency in the image while reducing computational complexity. For example, in a scene understanding task, superpixels might naturally group pixels belonging to the same object or surface, simplifying the subsequent analysis.

Object-level graphs

Object-level graphs represent images at a higher level of abstraction, with *nodes* corresponding to detected objects or regions of interest. *Edges* in these graphs often represent relationships or interactions between objects. This representation is particularly useful for tasks involving scene understanding, visual relationship detection, or high-level reasoning about image content.

Consider an image of a living room. An object-level graph might have *nodes* for "sofa," "coffee table," "lamp," and "bookshelf." *Edges* could represent spatial relationships (for example, "lamp on table") or functional relationships (for example, "person sitting on sofa"). Node features might include object class probabilities, bounding box coordinates, and appearance descriptors.

Object-level graphs are powerful for tasks that require reasoning about object interactions, such as visual question answering or image captioning. They allow the model to focus on relevant high-level information without getting bogged down in pixel-level details.

Scene graphs

Scene graphs take object-level representations a step further by explicitly modeling relationships between objects as separate entities in the graph. In a scene graph, *nodes* typically represent objects and attributes, while *edges* represent relationships. This structured representation captures the semantics of an image in a form that's closer to human understanding.

For example, in an image of a park, a scene graph might have nodes for "person," "dog," "tree," and "frisbee," with relationship edges such as "person throwing frisbee" or "dog under tree." Attributes such as "tree: green" or "frisbee: red" can be included as additional nodes or as node features.

Scene graphs are particularly valuable for tasks that require a deep understanding of image content, such as image retrieval based on complex queries, or generating detailed image descriptions. They provide a structured representation that bridges the gap between visual features and semantic understanding.

Comparing different graph construction methods

Each graph construction method has its strengths and is suited to different types of CV tasks. Pixel-level graphs preserve fine-grained information but can be computationally expensive. Superpixel graphs offer a good balance between detail and efficiency. Object-level and scene graphs capture high-level semantics but may miss fine-grained details.

The choice of graph construction method depends on the specific task, computational resources, and the level of abstraction required. For instance, image denoising might benefit from pixel-level graphs, while visual relationship detection would be better served by object-level or scene graphs.

It's also worth noting that these approaches aren't mutually exclusive. Some advanced models use hierarchical graph representations that combine multiple levels of abstraction, allowing them to reason about both fine-grained details and high-level semantics simultaneously.

The creation of graphs at different hierarchical levels (pixel, superpixel, and object) faces distinct challenges related to noise and data resolution. At the pixel level, high-frequency noise and sensor artifacts can create spurious connections, leading to unreliable graph structures. To address this, median filtering or bilateral filtering can be applied as preprocessing steps to preserve edges while reducing noise. Superpixel-level graphs encounter challenges with boundary precision and varying segment sizes, which can be mitigated through adaptive segmentation algorithms such as SLIC or using boundary refinement techniques.

Object-level graphs face resolution-dependent issues where object boundaries may be ambiguous or objects may appear at different scales. This can be addressed through multi-scale graph construction approaches or hierarchical graph representations that maintain connections across different resolution levels. To handle varying data resolutions, adaptive graph construction methods can be employed, where edge weights and neighborhood sizes are dynamically adjusted based on local data characteristics.

Another effective solution is to implement graph pooling strategies that aggregate information intelligently across different levels while preserving important structural relationships. Preprocessing techniques such as feature normalization and outlier removal can also improve graph quality. For cases with severe noise, weighted graph construction methods that incorporate uncertainty measures in edge weights have proven effective, allowing the model to learn more robust representations despite data imperfections.

Now, let's look at how exactly GNNs can be leveraged for image classification.

GNNs for image classification

Image classification, a fundamental task in CV, has traditionally been dominated by CNNs. However, GNNs are emerging as a powerful alternative, offering unique advantages in capturing global structure and long-range dependencies. This section will explore how GNNs can be applied to image classification tasks while discussing various architectures and techniques.

Graph convolutional networks for image data

Graph convolutional networks (**GCNs**) form the backbone of many graph-based approaches to image classification. Unlike traditional CNNs that operate on regular grid-like structures, GCNs can work with irregular graph structures, making them more flexible in representing image data.

To apply GCNs to images, we need to convert the image into a graph structure. This can be done using any of the methods discussed in the previous section, such as pixel-level graphs or superpixel graphs. Once we have the graph representation, we can apply graph convolutions to aggregate information from neighboring nodes.

For example, consider a superpixel-based graph of an image. Each node (superpixel) might have features such as average color, texture descriptors, and spatial information. A graph convolution operation would update each node's features based on its features and those of its neighbors. This allows the network to capture local patterns and gradually build up to more global representations.

Here's a simple example of how a graph convolution layer might be implemented using **PyTorch Geometric**:

```
import torch
from torch_geometric.nn import GCNConv

class SimpleGCN(torch.nn.Module):
    def __init__(self, num_node_features, num_classes):
        super(SimpleGCN, self).__init__()
        self.conv1 = GCNConv(num_node_features, 16)
        self.conv2 = GCNConv(16, num_classes)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        return x

# Usage
model = SimpleGCN(num_node_features=3, num_classes=10)
```

In this example, the model takes node features and an edge index as input, applies two graph convolution layers, and outputs class probabilities for each node. The final classification for the entire image could be obtained by pooling over all node predictions.

Attention mechanisms in graph-based image classification

Attention mechanisms have proven highly effective in various deep learning tasks, and they can be particularly powerful when applied to graph-based image classification. **Graph attention networks** (**GATs**) allow the model to assign different levels of importance to different neighbors when aggregating information, potentially leading to more effective feature learning.

In the context of image classification, attention can help the model focus on the most relevant parts of the image for the classification task. For instance, when classifying animal images, an attention mechanism might learn to focus on distinctive features such as the shape of the ears or the pattern of the fur, even if these features are spatially distant in the original image.

Consider an object-level graph representation of an image. An attention-based GNN could learn to assign higher importance to edges connecting objects that frequently co-occur in certain image classes. For example, in classifying "kitchen" scenes, the model might learn to pay more attention to edges connecting "stove" and "refrigerator" nodes as these objects are strongly indicative of kitchen environments.

Hierarchical graph representations for multi-scale feature learning

One of the strengths of CNNs in image classification is their ability to learn features at multiple scales through a hierarchy of convolutions and pooling operations. GNNs can achieve similar multi-scale feature learning through hierarchical graph representations.

A hierarchical graph approach might start with a fine-grained graph representation (for example, superpixel-level) and coarsen the graph progressively through pooling operations. Each level of the hierarchy captures features at a different scale, from local textures to more global shapes and arrangements.

For example, in classifying architectural styles, the lowest level of the hierarchy might capture local textures (brick patterns and window shapes), the middle levels might represent larger structures (roof types and facade layouts), and the highest levels could capture overall building shapes and arrangements.

This hierarchical approach can be implemented using graph pooling operations. Here's a conceptual example of how this might look in PyTorch Geometric:

```
from torch_geometric.nn import GCNConv, TopKPooling

class HierarchicalGCN(torch.nn.Module):
    def __init__(self, num_node_features, num_classes):
        super(HierarchicalGCN, self).__init__()
        self.conv1 = GCNConv(num_node_features, 64)
        self.pool1 = TopKPooling(64, ratio=0.8)
        self.conv2 = GCNConv(64, 32)
        self.pool2 = TopKPooling(32, ratio=0.8)
        self.conv3 = GCNConv(32, num_classes)

    def forward(self, x, edge_index, batch):
        x = self.conv1(x, edge_index)
        x, edge_index, _, batch, _, _ = self.pool1(
            x, edge_index, None, batch)
        x = self.conv2(x, edge_index)
        x, edge_index, _, batch, _, _ = self.pool2(
            x, edge_index, None, batch)
        x = self.conv3(x, edge_index)
        return x
```

In this example, the model applies alternating convolution and pooling operations, gradually reducing the graph size and capturing features at different scales.

GNNs versus CNNs

While GNNs offer several advantages for image classification, it's important to compare their performance with traditional CNN-based approaches. GNNs excel in capturing long-range dependencies and global structure, which can be beneficial for certain types of images and classification tasks. For instance, GNNs might outperform CNNs on tasks that require understanding the overall layout or relationships between distant parts of an image.

However, CNNs still hold advantages in their ability to capture local patterns efficiently and optimize for grid-like data. Many state-of-the-art approaches now combine elements of both GNNs and CNNs, leveraging the strengths of each.

For example, you might use a CNN to extract initial features from the image, then construct a graph based on these features and apply GNN layers for final classification. This approach combines the local feature extraction capabilities of CNNs with the global reasoning capabilities of GNNs.

In practice, the choice between GNN-based and CNN-based approaches (or a hybrid of the two) depends on the specific characteristics of the dataset and the nature of the classification task. Evaluating the target dataset empirically is often necessary to determine the most effective approach.

Object detection is one of the most important tasks in image understanding. Let's see how graphs can help us there.

Object detection and segmentation using GNNs

Object detection and **segmentation** are crucial tasks in CV, with applications ranging from autonomous driving to medical image analysis. While CNNs have been the go-to approach for these tasks, GNNs are emerging as a powerful alternative or complementary technique. This section will explore how GNNs can be applied to object detection and segmentation tasks while discussing various approaches and their advantages.

Graph-based object proposal generation

Object proposal generation is often the first step in many object detection pipelines. Traditional methods rely on sliding windows or region proposal networks, but graph-based approaches offer an interesting alternative. By representing an image as a graph, we can leverage the relational inductive bias of GNNs to generate more informed object proposals.

For example, consider an image represented as a graph of superpixels. Each superpixel (*node*) might have features such as color histograms, texture descriptors, and spatial information. *Edges* could represent adjacency or similarity between superpixels. A GNN can then process this graph to identify regions likely to contain objects.

Here's a simplified example of how a GNN might be used for object proposal generation:

```
import torch
from torch_geometric.nn import GCNConv, global_mean_pool

class ObjectProposalGNN(torch.nn.Module):
    def __init__(self, num_node_features):
        super(ObjectProposalGNN, self).__init__()
        self.conv1 = GCNConv(num_node_features, 64)
        self.conv2 = GCNConv(64, 32)
        self.conv3 = GCNConv(32, 1)  # Output objectness score

    def forward(self, x, edge_index, batch):
        x = torch.relu(self.conv1(x, edge_index))
        x = torch.relu(self.conv2(x, edge_index))
        x = self.conv3(x, edge_index)
        return x

# Usage
model = ObjectProposalGNN(num_node_features=10)
```

In this example, the model processes the graph and outputs an "objectness" score for each node (superpixel). These scores can then be used to generate bounding box proposals by grouping high-scoring adjacent superpixels.

Relational reasoning for object detection

One of the key advantages of using GNNs for object detection is their ability to perform relational reasoning. Objects in an image often have meaningful relationships with each other, and capturing these relationships can significantly improve detection accuracy.

For instance, in a street scene, knowing that a "wheel" object is next to a "car" object can increase the confidence of both detections. Similarly, detecting a "person" on a "horse" can help in classifying the scene as an equestrian event. GNNs can naturally model these relationships through message passing between object proposals.

Consider an approach where initial object proposals are generated (either through a traditional method or a graph-based approach, as discussed earlier), and then a GNN is used to refine these proposals:

```
class RelationalObjectDetectionGNN(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(RelationalObjectDetectionGNN, self).__init__()
        self.conv1 = GCNConv(num_features, 64)
        self.conv2 = GCNConv(64, 32)
        self.classifier = torch.nn.Linear(32, num_classes)
        self.bbox_regressor = torch.nn.Linear(32, 4)  # (x, y, w, h)

    def forward(self, x, edge_index):
        x = torch.relu(self.conv1(x, edge_index))
        x = torch.relu(self.conv2(x, edge_index))
        class_scores = self.classifier(x)
        bbox_refinement = self.bbox_regressor(x)
        return class_scores, bbox_refinement
```

In this model, each *node* represents an object proposal, and *edges* represent the relationships between proposals (for example, spatial proximity or feature similarity). The GNN refines the features of each proposal based on its relationships with other proposals, potentially leading to more accurate classifications and bounding box refinements.

Instance segmentation with GNNs

Instance segmentation, which combines object detection with pixel-level segmentation, can also benefit from graph-based approaches. GNNs can be used to refine segmentation masks by considering the relationships between different parts of an object or between different objects in the scene.

One approach is to represent an image as a graph of superpixels or pixels, where each node has features derived from a CNN backbone. A GNN can then process this graph to produce refined segmentation masks. This approach can be particularly effective for objects with complex shapes or in cases where global context is important for accurate segmentation.

For example, in medical image analysis, segmenting organs with complex shapes (such as the brain or lungs) can benefit from considering long-range dependencies and overall organ structure, which GNNs can capture effectively.

Here's a conceptual example of how a GNN might be used for instance segmentation:

```
class InstanceSegmentationGNN(torch.nn.Module):
    def __init__(self, num_features):
        super(InstanceSegmentationGNN, self).__init__()
        self.conv1 = GCNConv(num_features, 64)
        self.conv2 = GCNConv(64, 32)
        self.conv3 = GCNConv(32, 1) #Output per-node mask probability

    def forward(self, x, edge_index, batch):
        x = torch.relu(self.conv1(x, edge_index))
        x = torch.relu(self.conv2(x, edge_index))
        mask_prob = torch.sigmoid(self.conv3(x, edge_index))
        return mask_prob
```

This model takes a graph representation of an image (for example, superpixels) and outputs a mask probability for each node. These probabilities can then be used to construct the final instance segmentation masks.

Panoptic segmentation using graph-structured outputs

Panoptic segmentation, which aims to provide a unified segmentation of both **stuff** (amorphous regions such as sky or grass) and **things** (countable objects), presents a unique challenge that graph-based methods are well suited to address. GNNs can model the complex relationships between different segments in the image, whether they represent distinct objects or parts of the background.

A graph-structured output for panoptic segmentation might represent each segment (both stuff and things) as *nodes* in a graph. *Edges* in this graph could represent adjacency or semantic relationships between segments. This representation allows the model to reason about the overall scene structure and ensure consistency in the segmentation.

For instance, in a street scene, a graph-based panoptic segmentation model might learn that "car" segments are likely to be adjacent to "road" segments but not "sky" segments. This relational reasoning can help refine the boundaries between different segments and resolve ambiguities.

Here's a simplified example of how a GNN might be used for panoptic segmentation:

```
class PanopticSegmentationGNN(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super(PanopticSegmentationGNN, self).__init__()
        self.conv1 = GCNConv(num_features, 64)
        self.conv2 = GCNConv(64, 32)
        self.classifier = torch.nn.Linear(32, num_classes)
        self.instance_predictor = torch.nn.Linear(32, 1)

    def forward(self, x, edge_index):
        x = torch.relu(self.conv1(x, edge_index))
        x = torch.relu(self.conv2(x, edge_index))
        semantic_pred = self.classifier(x)
        instance_pred = self.instance_predictor(x)
        return semantic_pred, instance_pred
```

In this model, each node represents a segment in the image. The model outputs both semantic class predictions and instance predictions for each segment. The instance predictions can be used to distinguish between different instances of the same semantic class.

Next, we'll look at how to leverage GNNs to build intelligence over multiple modalities.

Multi-modal learning with GNNs

Multi-modal learning involves processing and relating information from multiple types of data sources or sensory inputs. In the context of CV, this often means combining visual data with other modalities such as text, audio, or sensor data. GNNs provide a powerful framework for multi-modal learning by naturally representing different types of data and their inter-relationships in a unified graph structure. This section will explore how GNNs can be applied to multi-modal learning tasks in CV.

Integrating visual and textual information using graphs

One of the most common multi-modal pairings in CV is the combination of visual and textual data. This integration is crucial for tasks such as image captioning, visual question answering, and text-based image retrieval. GNNs offer a natural way to represent and process these two modalities in a single framework.

For example, consider a visual question-answering task. We can construct a graph where nodes represent both image regions and words from the question. Edges can then represent relationships between these elements, such as spatial relationships between image regions or syntactic relationships between words. By applying graph convolutions to this heterogeneous graph, the model can reason about the relationships between the visual and textual elements to answer the question.

Here's a simplified example of how such a model might be structured. We begin with the necessary imports:

```
import torch
from torch_geometric.nn import GCNConv, global_mean_pool
```

The `VisualTextualGNN` class defines a visual-textual GNN model that can process both image and text data:

```
class VisualTextualGNN(torch.nn.Module):
    def __init__(self, image_feature_dim,
                 word_embedding_dim, hidden_dim):
        super(VisualTextualGNN, self).__init__()
        self.image_encoder = GCNConv(
            image_feature_dim, hidden_dim)
        self.text_encoder = GCNConv(
            word_embedding_dim, hidden_dim)
        self.fusion_layer = GCNConv(hidden_dim, hidden_dim)
        self.output_layer = torch.nn.Linear(
            hidden_dim, 1)  # For binary questions
```

The `forward` method shows how the model processes the input data through various layers to produce the final output:

```
    def forward(self, image_features, word_embeddings, edge_index):
        image_enc = self.image_encoder(
            image_features, edge_index)

        text_enc = self.text_encoder(
            word_embeddings, edge_index)

        fused = self.fusion_layer(
            image_enc + text_enc, edge_index)

        return self.output_layer(fused)
```

In this example, the model processes both image regions and words using separate GCN layers and then fuses this information in a subsequent layer. This allows the model to capture cross-modal interactions and reason about the relationship between visual and textual elements.

Cross-modal retrieval using graph-based representations

Cross-modal retrieval tasks, such as finding images that match a text description or vice versa, can benefit greatly from graph-based representations. GNNs can learn the joint embeddings of different modalities, allowing for efficient and accurate retrieval across modalities.

For instance, we could create a graph where nodes represent both images and text descriptions. Edges in this graph might connect similar images, similar text descriptions, and images with their corresponding descriptions. By applying GNN layers to this graph, we can learn embeddings that capture both intra-modal and cross-modal relationships.

Here's an example of how a GNN-based cross-modal retrieval model might be structured:

```
class CrossModalRetrievalGNN(nn.Module):
    def __init__(self, image_dim, text_dim, hidden_dim):
        super(CrossModalRetrievalGNN, self).__init__()
        self.image_encoder = GCNConv(image_dim, hidden_dim)
        self.text_encoder = GCNConv(text_dim, hidden_dim)
        self.fusion = GCNConv(hidden_dim, hidden_dim)

    def forward(self, image_features, text_features, edge_index):
        img_enc = self.image_encoder(image_features, edge_index)
        text_enc = self.text_encoder(text_features, edge_index)
        fused = self.fusion(img_enc + text_enc, edge_index)
        return fused
```

In this model, both images and text are encoded into a shared embedding space. The fusion layer allows for information to flow between the modalities, helping to align the embeddings. During retrieval, we can use these embeddings to find the nearest neighbors across modalities.

GNNs for visual-language navigation

Visual-language navigation is a complex task that requires understanding and integrating visual scene information with natural language instructions. GNNs can be particularly effective for this task by representing the navigation environment as a graph and incorporating language information into this graph structure.

For example, we could represent a navigation environment as a graph where nodes correspond to locations and edges represent possible movements between locations. Each node could have associated visual features extracted from images of that location. The natural language instructions can be incorporated by adding additional nodes or node features representing key elements of the instructions.

Here's a conceptual example of how a GNN might be used for visual-language navigation:

```
class VisualLanguageNavigationGNN(nn.Module):
    def __init__(self, visual_dim, instruction_dim,
                 hidden_dim, num_actions=4):
        super(VisualLanguageNavigationGNN, self).__init__()
        self.visual_gnn = GCNConv(visual_dim, hidden_dim)
        self.instruction_gnn = GCNConv(
            instruction_dim, hidden_dim)
        self.navigation_head = nn.Linear(
            hidden_dim * 2, num_actions)

    def forward(self, visual_obs, instructions,
                scene_graph, instr_graph):
        visual_feat = self.visual_gnn(visual_obs, scene_graph)
        instr_feat = self.instruction_gnn(
            instructions, instr_graph)
        combined = torch.cat([visual_feat, instr_feat], dim=-1)
        action_logits = self.navigation_head(combined)
        return action_logits
```

In this model, both visual scene information and language instructions are encoded and fused using GNN layers. The fused representations are then used to predict the next action in the navigation sequence.

Multi-modal learning with GNNs opens up exciting possibilities for more sophisticated and context-aware CV systems. By representing different modalities and their relationships in a unified graph structure, GNNs can capture complex interactions between modalities that are difficult to model with traditional approaches. This can lead to more robust and interpretable models for tasks that require information to be integrated from multiple sources.

As research in this area continues to advance, we can expect to see further innovations in graph-based architectures for multi-modal learning, potentially leading to breakthroughs in areas such as embodied AI, human-robot interaction, and advanced content retrieval systems.

It's important to understand the current challenges of performing graph-based learning on CV tasks. Let's go through some of them.

Limitations and next steps

As graph deep learning continues to make strides in CV, several challenges and promising research directions have begun to emerge. One of the primary challenges in applying graph-based methods to CV is scalability.

Scalability issues in large-scale visual datasets

As we saw in *Chapter 5*, as the size of visual datasets continues to grow, constructing and processing large graphs becomes computationally expensive. For instance, a high-resolution image represented as a pixel-level graph could contain millions of nodes, making it challenging to perform graph convolutions efficiently.

Researchers are exploring various approaches to address this issue. One promising direction is the development of more efficient graph convolution operations. For example, the **GraphSAGE** algorithm can be used with a sampling-based approach to reduce the computational complexity of graph convolutions. Another approach is to use **hierarchical graph representations**, where the graph is progressively coarsened, allowing for efficient processing of large-scale data.

Consider the following example of a hierarchical GNN that could be used to process large images:

```
class HierarchicalImageGNN(nn.Module):
    def __init__(self, input_dim, hidden_dims=[64, 32, 16]):
        super(HierarchicalImageGNN, self).__init__()
        self.levels = len(hidden_dims)
        self.gnns = nn.ModuleList()
        self.pools = nn.ModuleList()

        curr_dim = input_dim
        for hidden_dim in hidden_dims:
            self.gnns.append(GCNConv(curr_dim, hidden_dim))
            self.pools.append(TopKPooling(hidden_dim, ratio=0.5))
            curr_dim = hidden_dim

    def forward(self, x, edge_index, batch):
        features = []
        for i in range(self.levels):
            x = self.gnns[i](x, edge_index)
            x, edge_index, _, batch, _, _ = self.pools[i](
                x, edge_index, None, batch)
            features.append(x)
        return features
```

This model progressively reduces the graph's size, allowing it to process larger initial graphs more efficiently.

Efficient graph construction and updating for real-time applications

Many CV applications, such as autonomous driving or augmented reality, require real-time processing. Constructing and updating graphs on the fly for these applications presents a significant challenge. Future research needs to focus on developing methods for rapid graph construction and updating graph structures efficiently as new visual information becomes available.

One potential approach is to develop incremental graph construction methods that can update an existing graph structure with new information efficiently, rather than rebuilding the entire graph from scratch. For example, in a video processing task, we might want to update our scene graph as new frames arrive. Consider an autonomous vehicle navigating urban traffic. The system needs to maintain a dynamic scene graph that represents relationships between various objects, such as vehicles, pedestrians, traffic signs, and road infrastructure. As new frames arrive at 30 **frames per second** (**FPS**), the system must update this graph structure efficiently without compromising real-time performance.

For instance, when a new vehicle enters the scene, instead of reconstructing the entire graph, an incremental approach would only add new nodes and edges representing the vehicle and its relationships with existing objects. If a pedestrian moves from one location to another, only the edges representing spatial relationships need to be updated, while the core node attributes remain unchanged. This selective updating significantly reduces computational overhead compared to full graph reconstruction.

The system could employ a hierarchical graph structure where high-level relationships (such as vehicle-to-vehicle interactions) are updated less frequently than low-level details (such as precise object positions). This multi-scale approach allows for efficient resource allocation while maintaining accuracy where it matters most. For example, the relative positions of parked cars might be updated every few frames, while the trajectory of a crossing pedestrian requires frame-by-frame precision.

To further optimize performance, the system could implement a priority-based updating mechanism. Objects closer to the vehicle or those moving at higher speeds would receive more frequent updates than distant or stationary objects. This approach could be complemented with predictive models that anticipate object movements and pre-compute likely graph updates, reducing the processing load when new frames arrive.

Advanced data structures, such as spatial indices and efficient memory management schemes, can be employed to speed up node and edge updates. For instance, using R-trees or octrees to organize spatial information can significantly reduce the time needed to locate and update relevant graph components. Additionally, maintaining a cache of recently modified graph regions can help optimize frequent updates to dynamic parts of the scene.

These optimization strategies must be carefully balanced with memory constraints and the need to maintain graph consistency. The system should also be robust enough to handle edge cases, such as sudden changes in lighting conditions or occlusions, which may temporarily affect the quality of the visual information that's available for graph updates.

Integrating graph-based methods with other deep learning approaches

While graph-based methods offer unique advantages for CV tasks, they aren't a replacement for other deep learning techniques. Rather, the future likely lies in integrating graph-based methods with other approaches such as CNNs, transformers, and traditional CV algorithms effectively. For instance, we might use CNNs to extract initial features from images, construct a graph based on these features, and then apply GNN layers for further processing.

New applications and research opportunities

As graph deep learning for CV matures, new applications and research opportunities continue to emerge. Here are some exciting areas for future research:

- **Graph-based, few-shot, and zero-shot learning**: Leveraging graph structures to improve generalization to new classes with limited or no examples
- **Explainable AI through graph visualizations**: Using graph structures to provide more interpretable explanations of model decisions
- **Graph-based 3D vision**: Applying GNNs to 3D point cloud data for tasks such as 3D object detection and segmentation
- **Dynamic graph learning for video understanding**: Developing methods to learn and update graph structures over time for video analysis tasks
- **Graph-based visual reasoning**: Using GNNs to perform complex reasoning tasks on visual data, such as solving visual puzzles or answering multi-step visual questions

As these areas develop, we can expect to see new architectures, training methods, and theoretical insights that will further advance the field of graph deep learning for CV.

Summary

Graph deep learning has emerged as a powerful paradigm in CV, offering unique advantages in capturing relational information and global context across various tasks, from image classification to multi-modal learning. In this chapter, we've shown that by providing a more structured and flexible approach to visual data processing, graph-based methods address the limitations of traditional CNN-based approaches, excel at modeling non-grid structured data, and enhance the integration of multi-modal information.

You learned that as the field evolves, graph deep learning is poised to significantly impact real-world applications such as autonomous driving, medical imaging, augmented reality, robotics, and content retrieval systems. While challenges remain, particularly in scalability and real-time processing, the synergy between graph theory and deep learning promises to shape the future of CV, pushing toward more sophisticated visual reasoning and human-level understanding.

In the following chapter, we'll explore applications of graph learning beyond natural language processing, CV, and recommendation systems.

Part 4: Future Directions

In the final part of the book, you will discover additional applications of graph learning beyond the core domains and explore future directions. You will learn about the latest contemporary applications and gain insights into the challenges and opportunities that lie ahead in the field of graph learning.

This part has the following chapters:

11
Emerging Applications

Graph deep learning has demonstrated remarkable versatility across a wide array of domains, extending far beyond its well-known applications in **natural language processing** (**NLP**), recommendation systems, and **computer vision** (**CV**), as we saw in *Chapters 8*, *9*, and *10*, respectively. Here, we explore the diverse landscape of applications where graph-based approaches have made significant impacts or show promising potential.

As we delve into these applications, we'll see how graph deep learning techniques adapt to different contexts, often providing novel solutions to long-standing challenges. In urban planning, for example, these methods have been used to optimize public transportation networks and predict traffic flow, contributing to the development of smarter, more efficient cities. In the realm of materials science, researchers are leveraging graph-based models to predict material properties and design new compounds with specific characteristics, potentially accelerating innovation in fields such as renewable energy and advanced manufacturing.

The applications that we'll examine in this chapter not only showcase the breadth of graph deep learning's impact but also highlight the transferable nature of these techniques. Many of the approaches developed for one domain often find unexpected applications in others, fostering cross-pollination of ideas and methodologies. As we cover each application, we'll discuss the unique challenges they present, the specific graph deep learning techniques employed, and the results achieved.

By the end of this chapter, you'll have a comprehensive understanding of the far-reaching implications of graph deep learning across various industries and scientific disciplines, including the following:

- Biology and healthcare
- Social network analysis
- Financial services
- Cybersecurity
- Energy systems
- **Internet of Things** (**IoT**)
- Legal governance and compliance

Biology and healthcare

Graph-structured data is ubiquitous in biology and healthcare, from molecular interactions to brain connectomes. **Graph neural networks** (**GNNs**) have emerged as powerful tools for learning on these complex relational structures.

Protein-protein interaction networks

Protein-protein interaction (**PPI**) networks represent physical contact between proteins in a cell. These interactions are crucial for understanding cellular processes and developing new therapeutics. GNNs can effectively model and analyze PPI networks to do the following:

- **Predict new interactions**: GNNs can learn patterns in known interactions to infer novel PPIs. For example, Gainza et al. (2020) (`https://doi.org/10.1093/bioinformatics/btab154`) developed a GNN model that predicts PPIs by learning geometric and chemical features of protein surfaces. Another instance is the **Subgraph Neural Networks for Link Prediction (SEAL) model** by Zhang and Chen (2018) (`https://arxiv.org/pdf/1802.09691`), which achieved state-of-the-art performance in PPI prediction on the **Human Protein Reference Database** (**HPRD**). It learns from local enclosing subgraphs around protein pairs to predict interactions.
- **Identify functional modules**: By clustering proteins based on learned embeddings, GNNs can discover functional protein complexes. Xing et al. (`https://doi.org/10.1093/bioinformatics/btac088`) used a **graph attention network** (**GAT**) to identify disease-related protein modules.
- **Predict protein functions**: GNNs can propagate known functional annotations through PPI networks to predict functions of unannotated proteins. For instance, the **DeepGOPlus model** by Kulmanov and Hoehndorf (`https://doi.org/10.1093/bioinformatics/btaa763`) combines sequence information with PPI networks for improved function prediction.

Drug discovery and development

Graph-based models are revolutionizing various stages of the drug discovery pipeline:

- **Molecular property prediction**: GNNs can learn from molecular graphs to predict properties such as solubility, toxicity, and binding affinity. The **message passing neural network** (**MPNN**) by Gilmer et al. (`https://arxiv.org/abs/1704.01212`) pioneered this approach.
- **De novo drug design**: Generative GNN models can create novel molecular structures with desired properties. For example, the **GraphAF model** by Shi et al. (`https://arxiv.org/abs/2001.09382`) uses a flow-based approach to generate molecules atom by atom.

- **Drug-target interaction prediction**: GNNs can model both drugs and proteins as graphs to predict their interactions. The work by Nguyen et al. (`https://doi.org/10.1093/bioinformatics/btaa921`) uses the **GraphDTA model** for this task.
- **Polypharmacy side-effect prediction**: GNNs can model drug-drug interactions in knowledge graphs to predict the adverse effects of drug combinations. The **Decagon model** by Zitnik et al. (`https://doi.org/10.1093/bioinformatics/bty294`) is a prominent example.

Disease prediction and progression modeling

Graph-based models can integrate diverse biomedical data for improved disease prediction and understanding:

- **Electronic Health Record (EHR) analysis**: Patient records can be modeled as temporal graphs, with GNNs capturing complex relationships between diagnoses, medications, and lab tests. The **GRAM model** by Choi et al. (`https://pubmed.ncbi.nlm.nih.gov/33717639`) uses a graph-based attention mechanism on medical ontologies to improve risk prediction.
- **Disease gene prediction**: GNNs can integrate protein interactions, gene expression, and known disease associations to identify novel disease genes. The model by Li et al. (`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC8275341`) uses a heterogeneous **graph convolutional network** (**GCN**) for this task.
- **Cancer subtype classification**: Graph models can integrate multi-omics data (for example, gene expression, mutations, copy number variations) to improve cancer subtype prediction. The work of Zhang et al. (`https://doi.org/10.1016/j.engappai.2023.106717`) uses a multi-view GCN for this purpose.

Brain connectomics analysis

GNNs are particularly well suited for analyzing brain connectivity data:

- **Brain disorder classification**: GNNs can learn from structural or functional connectomes to classify neurological and psychiatric disorders. The **BrainGNN model** by Li et al. (`https://doi.org/10.1016/j.media.2021.102233`) uses a hierarchical GNN for autism spectrum disorder classification.
- **Cognitive state prediction**: Graph models can map brain connectivity patterns to cognitive states or behaviors. The work of Wang et al. (`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7935029`) uses a dynamic GCN to predict cognitive states from **functional magnetic resonance imaging** (**fMRI**) data.

Genomics and gene regulatory networks

Graph deep learning approaches are advancing our understanding of gene regulation and genomic architecture:

- **Enhancer-promoter interaction prediction**: GNNs can model the 3D structure of chromatin to predict long-range regulatory interactions. The model by Sun et al. (`https://genomebiology.biomedcentral.com/articles/10.1186/s13059-023-02916-x`) uses GCNs on Hi-C data for this task.
- **Variant effect prediction**: GNNs can model the impact of genetic variants on gene regulation and phenotypes. The **ExPecto model** by Zhou et al. (`https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6094955`) uses a deep learning approach that incorporates regulatory grammar to predict variant effects on gene expression.

Graph deep learning methods are making significant impacts across various domains in biology and healthcare. By effectively modeling complex relational data, these approaches are advancing our understanding of biological systems and improving clinical decision-making.

Social network analysis

Social networks provide rich relational data that can be naturally represented as graphs, with users as *nodes* and connections between users as *edges*. Graph deep learning techniques have emerged as powerful tools for analyzing and extracting insights from social network data.

Community detection

Community detection aims to identify clusters or groups of densely connected users within a social network, as shown in *Figure 11.1*. Traditional community detection algorithms, such as modularity optimization or spectral clustering, often struggle with large-scale networks. GNNs offer a promising alternative by learning node embeddings that capture both local and global network structures:

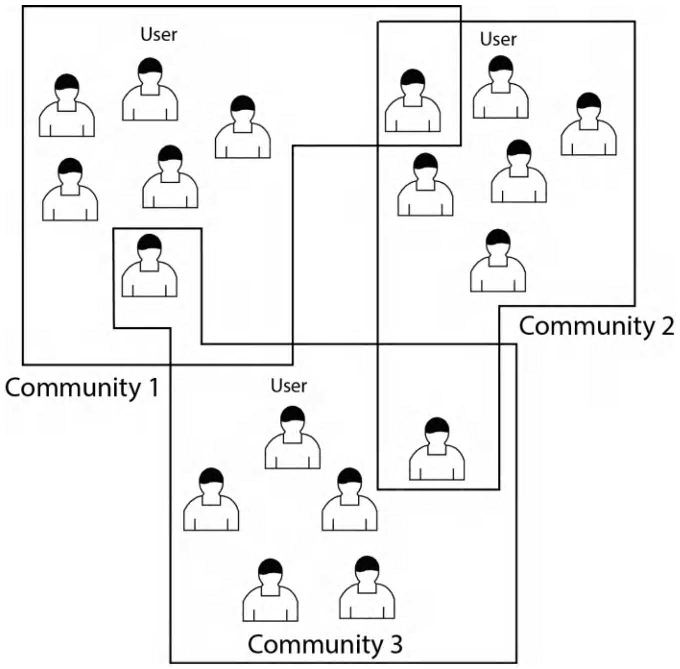


Figure 11.1 – Community detection

For instance, Wang et al. (2024) (`https://doi.org/10.1016/j.neucom.2024.127703`) proposed a **graph autoencoder** (**GAE**) approach for community detection. The model uses graph convolutional layers to encode nodes into a low-dimensional latent space, then attempts to reconstruct the graph structure from these embeddings. By training on the reconstruction task, the embeddings naturally cluster nodes belonging to the same community. The decoder can then be used to assign community labels.

Influence propagation modeling

Understanding how information and influence spread through social networks is crucial for applications such as viral marketing and public health campaigns. GNNs can model complex diffusion dynamics by learning how node attributes and network structure impact propagation patterns.

The **DeepInf model** by Qiu et al. (`https://dl.acm.org/doi/10.1145/3219819.3220077`) uses a GAT to predict whether a user will be influenced to adopt a behavior based on their local network neighborhood. The attention mechanism allows the model to focus on the most influential neighbors when making predictions.

User behavior prediction

Predicting user behaviors and preferences is a key task for recommender systems and targeted advertising. GNNs can improve predictions by incorporating social context and network effects.

For example, the **GraphRec model** by Fan et al. (`https://arxiv.org/abs/1902.07243`) uses a GCN to learn user and item embeddings from a user-item interaction graph for social recommendation. The model also incorporates a social aggregation layer to capture influence from a user's social connections.

Fake news detection

The spread of misinformation on social media has become a major societal challenge. Graph-based deep learning models can help detect fake news by analyzing both content features and propagation patterns.

For instance, the **GCAN model** by Lu and Li (`https://aclanthology.org/2020.acl-main.48`) uses a graph-aware **co-attention network** (**CAN**) to jointly model news content and user engagement patterns. The model learns representations of news articles, user profiles, and user-news interactions to make predictions.

By capturing complex relational structures and learning rich node representations, GNNs can improve performance on a wide range of social network analysis tasks.

Financial services

Graph-based deep learning models have emerged as powerful tools for analyzing complex relationships and patterns in financial data. By representing financial entities and their interactions as nodes and edges in a graph, these models can capture intricate dependencies that traditional methods often miss.

Fraud detection in transaction networks

One of the most impactful applications of graph-based deep learning in finance is detecting fraudulent activities in transaction networks. Traditional fraud detection systems often rely on rule-based approaches or analyze transactions in isolation. However, fraudulent behaviors frequently involve complex patterns of interactions between multiple entities over time.

GNNs can model entire transaction networks, where nodes represent accounts or users, and edges represent transactions or relationships between entities. By propagating and aggregating information across the graph structure, GNNs can identify suspicious patterns that may be invisible when looking at individual transactions alone.

For instance, a team at Alibaba developed a GNN-based fraud detection system called **GraphConsis** that significantly outperformed traditional methods. The model represents users, merchants, and devices as nodes in a heterogeneous graph, with edges representing various types of interactions. By learning node embeddings that capture both local and global graph structures, GraphConsis can identify coordinated fraud rings and detect subtle anomalies in transaction patterns.

Credit risk assessment

Assessing creditworthiness is a critical task for financial institutions. Graph-based models can enhance credit scoring by incorporating diverse data sources and modeling complex relationships between borrowers, their financial activities, and external factors.

For example, researchers have proposed multiple graph-based credit scoring models. These approaches construct a graph where *nodes* represent loan applicants and *edges* represent similarities between applicants based on various features. The models then use GCNs to propagate credit information across similar applicants, leading to more accurate and robust credit scores. Such models can also incorporate domain knowledge as constraints during the learning process, ensuring the model's predictions align with established financial principles.

Stock market prediction using company relationship graphs

The stock market is a complex system where companies' performances are often interrelated due to supply chain relationships, competition, and broader economic factors. Graph-based models can capture these intricate relationships to improve stock price prediction and portfolio management.

A study by researchers introduced the **long short-term memory graph convolutional neural network** (**LSTM-GCN**) model for stock market prediction (`https://arxiv.org/abs/2303.09406`). This model constructs a dynamic graph of companies, where edges represent value chain relationships and correlations in historical stock prices. The LSTM-GCN model then learns to aggregate information from neighboring companies and across time to predict future stock prices.

Anti-money laundering systems

Money laundering often involves complex networks of transactions designed to obscure the origin of illicit funds. Graph-based deep learning models are particularly well suited for **anti-money laundering** (**AML**) applications, as they can analyze entire transaction networks to identify suspicious patterns.

For example, researchers have developed a GNN-based AML system called **GCN-AML** that operates on Bitcoin transaction graphs (`https://arxiv.org/abs/1908.02591`). The model represents Bitcoin addresses as nodes and transactions as edges in a large-scale graph. By applying GCNs to this structure, GCN-AML learns to identify patterns indicative of money laundering activities, such as layering and integration of funds. The system demonstrated high accuracy in detecting known money laundering cases while also uncovering previously unidentified suspicious activities.

Personalized financial recommendations

Graph-based models can also enhance personalized financial services by modeling relationships between customers, products, and financial behaviors.

For instance, a major bank implemented a GNN-based recommendation system for personalized financial product offerings. The system constructs a heterogeneous graph incorporating *customer nodes*, *product nodes*, and various *interaction edges* (for example, past purchases, inquiries, and demographic similarities). By learning embeddings that capture both customer preferences and product characteristics, the GNN can generate highly relevant and personalized product recommendations, leading to increased customer satisfaction and product adoption rates.

Systemic risk assessment

Regulators and central banks are increasingly interested in using graph-based models to assess systemic risks in financial networks, particularly in the wake of the 2008 financial crisis.

For example, researchers have proposed using GNNs to model interbank lending networks and predict systemic risks. By representing banks as nodes and their lending relationships as edges, GNNs can learn to identify vulnerable institutions and potential contagion paths in the financial system. This approach allows for more dynamic and data-driven systemic risk assessments compared to traditional stress testing methods.

Graph-based deep learning models offer powerful tools for analyzing the complex, interconnected nature of financial systems. As these techniques continue to evolve, we can expect to see even more sophisticated applications that leverage the rich relational structure of financial data to improve decision-making, risk management, and customer experiences across the financial services industry.

Cybersecurity

GNNs have emerged as a powerful tool for cybersecurity applications, leveraging the inherent graph structure of many security-related datasets to detect threats and anomalies.

Why graphs for cybersecurity?

Many cybersecurity datasets and problems naturally lend themselves to graph representations:

- Network traffic and communications can be modeled as graphs, with devices as *nodes* and connections as *edges*.
- System call traces form temporal graphs of process interactions.
- Social networks used for fraud detection are inherently graph-structured.
- Software dependency graphs represent relationships between code components.

By using graph-based models, we can capture and analyze complex relationships and patterns that may be missed by traditional **machine learning** (**ML**) approaches.

Network intrusion detection

GNNs have shown promise for detecting network intrusions and anomalies by analyzing traffic patterns. In this application, the following points are relevant:

- *Nodes* represent devices or IP addresses.
- *Edges* represent network connections or data transfers.
- *Node features* may include device type, operating system, and so on.
- *Edge features* may include protocol, port numbers, data volume, and so on.

A GNN can learn to identify suspicious patterns of communication that may indicate an ongoing attack or compromised device. For example, a GNN-based **intrusion detection system** (**IDS**) might flag unusual data transfers between nodes that don't typically communicate or identify a node suddenly establishing many new connections:

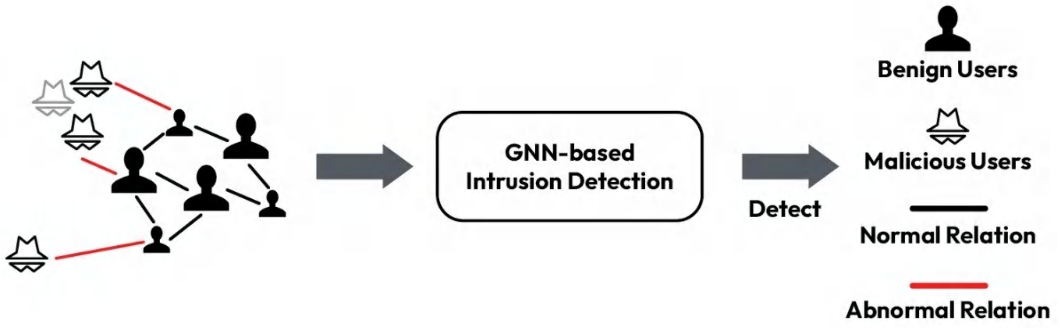


Figure 11.2 – Intrusion detection using graph learning

Malware detection

Graph representations of program behavior, such as system call graphs or control flow graphs, can be analyzed using GNNs to detect malicious software. In this case, note the following:

- *Nodes* represent system calls or code blocks.
- *Edges* represent temporal ordering or control flow.
- *Node features* may include call types, arguments, return values, and so on.

GNNs can learn to distinguish between benign and malicious behavioral patterns. For instance, a GNN might identify sequences of system calls characteristic of ransomware file encryption operations.

Let's take an example of **advanced persistent threat** (**APT**) detection. APTs are sophisticated, long-term cyber-attacks that are particularly challenging to detect. Let's consider a hypothetical GNN-based APT detection system:

1. **Graph construction**

- *Nodes* represent network entities (devices, users, IP addresses).
- *Edges* represent communications or access events.
- *Node features* include device types, installed software, and user roles.
- *Edge features* include communication protocols, data volumes, and timestamps.

  Let's look at an example:

```
def construct_graph(network_data):
    # devices, users, IP addresses
    nodes = create_nodes(network_data)
    # communications, access events
    edges = create_edges(network_data)

    for node in nodes:
        # device type, software, user role
        node.features = extract_node_features(node)

    for edge in edges:
        # protocol, data volume, timestamp
        edge.features = extract_edge_features(edge)

    return Graph(nodes, edges)
```

2. **GNN architecture**

- Use a GAT to learn node embeddings.
- *Temporal edges* are handled with a recurrent mechanism (for example, **gated recurrent unit** (**GRU**) cells).
- Multiple GNN layers capture multi-hop relationships.

  Here's how this could work:

```
class APTDetectionGNN(nn.Module):
    def __init__(self, in_features, hidden_features,
                 num_layers):
        super().__init__()
        self.gat_layers = nn.ModuleList([
            GATConv(in_features if i == 0 else hidden_features,
                    hidden_features
            )for i in range(num_layers)
        ])
        self.gru = nn.GRU(hidden_features, hidden_features)
        self.output = nn.Linear(hidden_features, 1)

    def forward(self, x, edge_index, edge_attr):
        for gat in self.gat_layers:
            x = F.relu(gat(x, edge_index, edge_attr))
        x, _ = self.gru(x.unsqueeze(0))
        return self.output(x.squeeze(0)).squeeze(-1)
```

3. **Training**

Here is how we will write a model training code for APT incident detection using historical labeled data.

```
  def train_model(model, labeled_data, unlabeled_data,
                  num_epochs=10):
      optimizer = torch.optim.Adam(model.parameters())

      for epoch in range(num_epochs):
          model.train()

          # Supervised learning
          optimizer.zero_grad()
          x, edge_index, edge_attr, y = labeled_data
          out = model(x, edge_index, edge_attr)
          loss = F.binary_cross_entropy_with_logits(out, y)
          loss.backward()
          optimizer.step()

          # Print training progress
          if epoch % 2 == 0:
              print(f"Epoch {epoch}, Loss: {loss.item():.4f}")
```

4. **Detection**

- The GNN outputs anomaly scores for each node.
- Nodes with high anomaly scores are flagged for investigation.
- Attention weights can be analyzed to explain which connections contributed to the anomaly score.

Let's take the following approach:

```
def detect_apts(model, graph, threshold=0.5):
    model.eval()
    with torch.no_grad():
        x, edge_index, edge_attr, _ = graph
        anomaly_scores = torch.sigmoid(model(
            x, edge_index, edge_attr))
        suspicious_nodes = (
            anomaly_scores > threshold
        ).nonzero().flatten()

        results = []
        for node in suspicious_nodes:
            attention_weights = model.get_attention_ weights(
                node)
            results.append({
                'node': node.item(),
                'score': anomaly_scores[node].item(),
                'attention': attention_weights
            })

        return results
```

This approach could detect subtle APT patterns by connecting seemingly unrelated events, such as unusual login times, atypical system access, and low-volume data exfiltration, into a coherent picture of an ongoing attack. The attention weights of the model could then be analyzed to explain which connections contributed to the anomaly score, providing valuable insights for security teams.

Energy systems

Graph-based deep learning models can act as powerful tools for analyzing and optimizing complex energy systems. By representing energy networks as graphs, these models can capture intricate relationships and dependencies between different components, enabling more accurate predictions, efficient control strategies, and improved decision-making.

Graph representation of energy systems

Energy systems can be naturally represented as graphs, where *nodes* typically represent physical components such as generators, transformers, transmission lines, and loads, while *edges* represent connections and interactions between these components. This graph structure allows us to model the following:

- Power flow between components
- Interdependencies in the network

- Spatial and temporal relationships
- System topology and connectivity

Load forecasting

Graph-based deep learning models have shown superior performance in predicting electricity demand across power networks. By incorporating the grid topology and spatial correlations between loads, these models can capture complex patterns more effectively than traditional time series approaches.

For example, a GCN-LSTM hybrid model can be used for short-term load forecasting in a city-wide power grid, where each *node* represents a substation and *edges* represent transmission lines. The model aggregates information from neighboring substations and historical load data to make accurate predictions.

Fault detection and localization

GNNs can analyze the propagation of anomalies through power networks, enabling rapid detection and localization of faults.

For instance, a GAT-based model can be used for identifying faulty components in a transmission network. The attention mechanism allows the model to focus on the most relevant neighboring nodes when detecting anomalies, improving accuracy and interpretability.

Optimal power flow

GNNs can learn to solve complex optimization problems such as **optimal power flow** (**OPF**) more efficiently than traditional methods.

For example, a GNN-based mode can be trained to approximate OPF solutions, taking into account network constraints and renewable energy uncertainties. The model can generate near-optimal solutions in milliseconds, enabling real-time optimization of power dispatch.

Renewable energy forecasting

Spatial-temporal GNNs (**STGNNs**) are particularly effective for forecasting renewable energy generation, as they can capture both spatial correlations (for example, between nearby wind farms) and temporal patterns.

For instance, an STGNN model can be used for predicting solar power generation across a network of **photovoltaic** (**PV**) installations. The model can incorporate weather data, historical generation, and spatial relationships between sites to improve forecast accuracy.

Energy storage management

Graph-based **reinforcement learning** (**RL**) models can optimize the operation of distributed energy storage systems, considering network constraints and market conditions.

For example, a **graph reinforcement learning** (**GRL**) agent can be used for coordinating a network of battery energy storage systems in a microgrid. The agent can learn to charge and discharge batteries optimally, balancing renewable integration, peak shaving, and electricity market participation.

Vulnerability assessment

GNNs can analyze the resilience of power grids to various disturbances and identify critical components.

For instance, a GCN-based model can be used for assessing the vulnerability of power grids to cascading failures. The model can learn to predict the propagation of failures through the network and identify the most critical nodes for system stability.

IoT

Graph-based deep learning models can be used for analyzing the complex interconnected nature of IoT systems. By representing IoT devices, sensors, and their interactions as *nodes* and *edges* in a graph structure, these models can capture important relational information and dependencies that are crucial for various IoT applications.

Device interaction modeling

One of the fundamental challenges in IoT systems is modeling the complex interactions between heterogeneous devices and sensors. GNNs provide an elegant solution to this problem by naturally representing devices as *nodes* and their communications or dependencies as *edges* in a graph.

GCNs have been particularly effective for this task. In a GCN, each node aggregates information from its neighbors through convolutional operations, allowing the model to learn representations that incorporate both node features and graph structure. This enables the GCN to capture complex interaction patterns between IoT devices.

For example, in a smart home setting, a GCN could model the relationships between various smart appliances, environmental sensors, and user devices. The model could learn how the activation of one device (for example, a motion sensor) influences the behavior of others (for example, lights or thermostats). This learned representation can then be used for tasks such as predicting device states or optimizing overall system performance.

Anomaly detection in sensor networks

Detecting anomalies in IoT sensor networks is crucial for identifying faults, security breaches, or unusual events. Graph-based deep learning models excel at this task by leveraging spatial and temporal correlations between sensor readings.

GATs have shown promising results in this domain. GATs use attention mechanisms to assign different weights to neighboring nodes, allowing the model to focus on the most relevant information for anomaly detection. This is particularly useful in large-scale IoT deployments where not all sensor interactions are equally important.

In practice, a GAT-based anomaly detection system might work as follows:

1. Represent sensors as *nodes* in a graph, with *edges* based on physical proximity or logical relationships.
2. Use historical sensor readings as *node features*.
3. Apply graph attention layers to learn representations that capture normal behavior patterns.
4. Use these learned representations to identify deviations that may indicate anomalies.

This approach has been successfully applied to detect anomalies in urban water distribution networks, **industrial control systems** (**ICSs**), and smart grid infrastructures.

Predictive maintenance

Predictive maintenance is a critical application in **industrial IoT** (**IIoT**) settings, where unexpected equipment failures can lead to significant costs and safety risks. Graph-based deep learning models can enhance predictive maintenance by incorporating complex interdependencies between different components of industrial systems.

Temporal GNNs (**TGNNs**) are particularly well suited for this task, as they can model both the spatial relationships between components and their temporal evolution. A TGNN for predictive maintenance might achieve the following:

1. Represent equipment components as *nodes* and their physical or functional connections as *edges*.
2. Use time series of sensor readings and operational data as dynamic *node features*.
3. Apply GCN and **recurrent neural network** (**RNN**) layers to capture both spatial and temporal patterns.
4. Predict future component states or failure probabilities.

This approach has been shown to outperform traditional ML methods in predicting equipment failures in manufacturing plants, wind turbines, and aircraft engines.

Smart home applications

In smart home environments, graph-based deep learning can enhance various applications by modeling complex interactions between devices, users, and the environment.

For example, a graph-based recommender system for smart home automation might use a heterogeneous graph structure as follows:

- Devices, users, and rooms are represented as different types of *nodes*.
- *Edges* represent interactions (for example, user-device interactions) or relationships (for example, device locations).
- *Node features* include device characteristics, user preferences, and environmental data.

By applying techniques such as graph embedding or graph convolutional matrix completion, such a system can generate personalized automation rules or device usage recommendations that account for the full context of the smart home ecosystem.

Legal governance and compliance

GNNs can be useful tools in the legal and compliance sectors, offering innovative solutions to complex challenges. By leveraging graph structures to represent intricate relationships between legal entities, documents, and regulations, these techniques are revolutionizing how legal professionals and compliance officers approach their work.

Knowledge graph construction for legal and regulatory data

One of the primary applications of deep learning on graphs in the legal domain is the construction and utilization of **knowledge graphs**. These graphs serve as comprehensive repositories of legal and regulatory information, capturing complex relationships between various entities such as laws, regulations, court cases, and legal concepts:

- **Entity extraction and relation mapping**: Deep learning models, particularly those based on NLP techniques, are employed to automatically extract entities and relationships from legal texts. GNNs can then be used to refine and expand these relationships, creating a rich, interconnected network of legal knowledge.
- **Multilingual integration**: In regions with multiple official languages, such as the **European Union** (**EU**), deep learning on graphs facilitates the integration of legal information across different languages. This enables the creation of multilingual legal knowledge graphs that support cross-border compliance and legal research.
- **Dynamic updating**: The legal landscape is constantly evolving. deep learning models can be trained to continuously update the knowledge graph with new legislation, case law, and regulatory changes, ensuring that the graph remains current and relevant.

Automated compliance monitoring and risk assessment

Deep learning on graphs offers powerful tools for automating compliance processes and assessing regulatory risks:

- **Pattern recognition in complex regulations**: GNNs can analyze the structure of regulatory texts represented as graphs, identifying patterns and relationships that might be missed by traditional text analysis methods. This capability is particularly useful for understanding the implications of new regulations across different business areas.
- **Real-time compliance checking**: By representing an organization's operations and regulatory requirements as a graph, deep learning models can perform real-time compliance checks. These models can flag potential violations and suggest corrective actions, significantly reducing the risk of non-compliance.
- **Predictive risk analysis**: By analyzing historical compliance data and current regulatory trends represented in graph form, deep learning models can predict future compliance risks. This allows organizations to proactively address potential issues before they become serious problems.

Legal document analysis and contract management

Deep learning on graphs is transforming how legal documents are analyzed and managed:

- **Semantic understanding of legal documents**: By representing legal documents as graphs of interconnected concepts and clauses, deep learning models can achieve a more nuanced understanding of document content. This enables more accurate document classification, comparison, and information retrieval.
- **Automated contract review**: GNNs can be trained to analyze contract graphs, identifying key clauses, potential risks, and inconsistencies. This significantly speeds up the contract review process and improves accuracy.
- **Legal precedent analysis**: By representing case law as a graph of interconnected decisions, deep learning models can analyze legal precedents more effectively. This aids in case preparation and predicting potential outcomes of legal disputes.

Regulatory intelligence and policy impact assessment

Deep learning on graphs is enhancing how organizations understand and respond to regulatory changes:

- **Regulatory change management**: By representing regulatory frameworks as graphs, deep learning models can analyze the impact of changes in one regulation on other related regulations. This helps organizations understand the full implications of regulatory changes and adjust their compliance strategies accordingly.

- **Policy impact prediction**: GNNs can be used to model the potential impacts of proposed regulations or policy changes. By analyzing the connections between different areas of law and business operations, these models can predict how new policies might affect various stakeholders.

Summary

In this chapter, we investigated the diverse applications of graph deep learning across various domains. You learned about how graph-based approaches are revolutionizing fields such as biology and healthcare, social network analysis, financial services, cybersecurity, energy systems, IoT, and legal governance and compliance.

We also highlighted how GNNs and other graph-based models can capture complex relationships in interconnected data structures, leading to breakthroughs in areas such as drug discovery, fraud detection, network optimization, and predictive maintenance. By showcasing the adaptability and transferability of these techniques, we explained the far-reaching impact of graph deep learning in solving complex real-world problems across industries and scientific disciplines.

In the next chapter, we will look at what lies in the future for graph deep learning and which areas will be deeply impacted by these powerful models.

12
The Future of Graph Learning

Throughout this book, we've covered a wide range of topics regarding graph learning, from fundamental concepts to cutting-edge applications. You now have a solid foundation to tackle complex graph-based problems and contribute to this rapidly evolving field.

The field of graph learning stands at the cusp of a revolutionary era, poised to transform how we understand and interact with complex, interconnected data. As we look ahead, several key trends and advancements are shaping the trajectory of this dynamic field.

In this last chapter, we'll discuss the following topics:

- Emerging trends and directions
- Advanced architectures and techniques
- Integration with other **artificial intelligence** (**AI**) domains
- Potential breakthroughs and long-term vision

Emerging trends and directions

The new trends in graph learning reflect both the growing capabilities of graph-based models and the expanding range of applications where they're being deployed. From advances in model architectures to novel training techniques, the following developments are at the forefront of graph learning research and practice.

Scalability and efficiency

As we saw in *Chapter 5*, the ability to handle increasingly large and complex graphs is becoming a crucial challenge as data volumes grow exponentially. Researchers are developing innovative approaches to tackle this challenge.

Handling larger and more complex graphs

New algorithms are being designed to process graphs with billions of nodes and edges efficiently (for more details on node- and edge-level learning, please refer to *Chapter 2*). These methods often leverage the sparsity and locality properties of real-world graphs. For example, sampling-based approaches such as **GraphSAGE** (see *Chapter 4*) and **FastGCN** have shown promise in scaling **graph neural networks** (**GNNs**) to large graphs by operating on subsets of nodes rather than the entire graph. Another direction is the development of more efficient aggregation schemes, such as the **Cluster-GCN** method (also discussed in *Chapter 4*), which pre-processes the graph into smaller clusters to reduce computational complexity.

Distributed and parallel graph learning algorithms

Techniques such as graph partitioning and distributed training allow massive graphs to be processed across multiple machines or **graph processing units** (**GPUs**). This allows us to scale to previously intractable problem sizes. Distributed GNN frameworks such as **DistDGL** (`https://arxiv.org/abs/2010.05337`) and **AliGraph** (`https://arxiv.org/abs/1902.08730`) are making it possible to train models on graphs with billions of nodes and edges. These systems often employ sophisticated partitioning strategies to minimize communication overhead while maintaining model accuracy.

Graph compression techniques

Novel methods for compressing graph structures while preserving important topological information are emerging. These techniques reduce memory requirements and computational complexity, making it feasible to work with enormous graphs on limited hardware. Approaches such as **graph sparsification** and **node pruning** can significantly reduce graph size while maintaining essential structural properties. Additionally, techniques such as quantization and low-rank approximation are being applied to GNN models to reduce their memory footprint.

Interpretability and explainability

As graph learning models become more complex, the need for interpretability grows.

Developing methods to understand GNN decisions

Researchers are creating techniques to visualize and explain the decision-making process of GNNs. This includes methods such as attention visualization and feature importance analysis. For instance, **GNNExplainer** provides a way to identify important subgraphs and features for individual predictions. Other approaches, such as **PGExplainer**, focus on generating human-readable explanations for GNN predictions.

Visualization techniques for graph learning models

Advanced visualization tools are being developed to help researchers and practitioners understand the inner workings of graph models. These tools can reveal patterns in node embeddings (which we also explored in *Chapter 3*), highlight important subgraphs, and show how information flows through the graph. Projects such as **GraphViz** and **NetworkX** are being extended to support the visualization of GNN architectures and their learned representations.

Causal inference in graph structures

There's growing interest in understanding causal relationships within graphs. This involves developing methods to distinguish between correlation and causation in graph data, which is crucial for many real-world applications. Causal discovery algorithms for graphs such as **directed acyclic graph-graph neural networks** (**DAG-GNNs**) are being developed to infer causal structures from observational data. These methods have potential applications in fields such as healthcare and social sciences.

Dynamic and temporal graphs

Many real-world graphs evolve, necessitating new approaches.

Learning about evolving graph structures

Techniques are being developed to handle graphs where nodes and edges appear or disappear over time. This is particularly important for applications such as social network analysis and financial fraud detection. Models such as **EvolveGCN** and **DynGEM** can update node representations efficiently as the graph structure changes. These approaches often use recurrent architectures to capture temporal dependencies.

Incorporating temporal information in graph models

Researchers are exploring ways to embed time-related information directly into graph models. This allows complex temporal dependencies and patterns to be captured in dynamic graphs. **Temporal graph neural networks** (**TGNNs**) (see *Chapter 11*) and **time-aware graph neural networks** (**TA-GNNs**) are examples of architectures that are designed to handle continuous-time dynamic graphs.

Predicting future graph states

Advanced models are being created to forecast how graphs will evolve. This has applications in areas such as traffic prediction, epidemic modeling, and recommendation systems (which we looked at in detail in *Chapter 9*). Methods such as **Graph WaveNet**, **spatial-temporal graph neural networks** (**STGNNs**) (see *Chapter 11*), and **spatial-temporal graph convolutional networks** (**STGCNs**) combine graph convolutions with temporal convolutions to capture both spatial and temporal dependencies for forecasting tasks.

Heterogeneous and multi-modal graphs

Real-world graphs often contain diverse types of nodes and edges, as well as multiple data modalities.

Handling diverse node and edge types

New architectures are being designed to process graphs with heterogeneous node and edge types effectively. This is crucial for applications such as knowledge graphs and biological networks. **Heterogeneous graph neural networks** (**HGNNs**) (see *Chapter 4*) and **relational graph convolutional networks** (**R-GCNs**) are examples of models that can handle multiple node and edge types.

Integrating multiple data modalities with graphs

Researchers are developing methods to combine graph data with other modalities such as text, images, and audio. This allows for richer representations and more powerful models. For instance, **Graph-BERT** combines graph structure with textual information using transformer architectures. In the field of computer vision, which we covered in *Chapter 10*, methods such as STGNNs integrate visual and graph data for tasks such as action recognition.

Cross-modal learning on graphs

Techniques for transferring knowledge between different modalities within a graph are emerging. This enables more robust and versatile graph learning models. Approaches such as **graph cross-modal attention networks** allow information to be exchanged between different modalities, enhancing performance on tasks that require multi-modal reasoning.

Advanced architectures and techniques

From advanced transformer architectures to cutting-edge generative models and innovative reinforcement learning strategies, graph learning demonstrates immense potential across a diverse set of tasks.

Graph transformers and attention mechanisms

The success of transformer architectures in **natural language processing** (**NLP**), which we looked at in *Chapter 8*, is inspiring new approaches in graph learning.

Adapting transformer architectures for graph data

Researchers are modifying transformer models so that they work effectively with graph-structured data. This allows long-range dependencies and global context to be captured in graphs. **Graph transformer networks** (**GTNs**) adapt the self-attention mechanism to operate on graph-structured data, enabling the model to learn complex relationships between nodes. These models can dynamically adjust the graph structure during the learning process, potentially discovering hidden relationships that are not explicitly present in the original graph.

Self-attention mechanisms for graph learning

Novel attention mechanisms designed specifically for graphs are being developed. These allow models to focus on the most relevant parts of a graph for a given task. **Graph attention networks** (**GATs**) (see *Chapter 4*) introduce attention coefficients to weigh the importance of different neighboring nodes during the aggregation step. This enables the model to assign different importance to different nodes, improving performance on various graph-based tasks.

Long-range dependencies in graphs

New techniques are emerging to capture relationships between distant nodes in a graph efficiently. This is particularly important for tasks that require global graph structure to be understood. Methods such as **adaptive graph convolutional networks** (**AGCNs**) and graph wavelets are being developed to capture multi-scale information in graphs, allowing models to consider both local and global graph structures simultaneously.

AGCNs are designed to dynamically adjust the graph structure during the learning process. This adaptive nature allows them to capture and model relationships between nodes that may not be directly connected in the original graph structure. By doing so, AGCNs can establish connections between distant nodes effectively, thus addressing the long-range dependency problem that traditional GCNs often struggle with.

Graph generative models

The ability to generate realistic graph structures is opening up new possibilities.

Generating realistic graph structures

Advanced generative models, such as **graph variational autoencoders** (**GVAEs**) and **graph generative adversarial networks** (**GraphGANs**), are being developed to create synthetic graphs that mimic real-world network properties. These models can learn to generate graphs with specific characteristics, such as degree distribution, clustering coefficient, and community structure. This capability is particularly useful for simulating complex systems and generating benchmark datasets.

Applications in drug discovery and material science

Graph generative models are being used to design new molecules and materials with desired properties, potentially accelerating scientific discovery. Models such as **MolGAN** and **GraphAF** can generate novel molecular structures with specific chemical properties, aiding in the discovery of new drugs and materials. These approaches have the potential to significantly reduce the time and cost associated with traditional experimental methods in these fields.

GVAEs and GANs

These models are being refined to generate increasingly realistic and diverse graph structures, with applications ranging from social network simulation to protein design. Recent advancements include conditional graph generation, where models can generate graphs with specific properties or constraints. This has applications in areas such as network design, where graphs with certain structural properties are desired.

Few-shot and zero-shot learning on graphs

Few-shot learning on graphs refers to the ability of a model to learn and make predictions on graph-structured data with only a small number of labeled examples. This approach is particularly useful when dealing with large-scale graph datasets where obtaining labeled data is expensive or time-consuming.

Zero-shot learning on graphs, on the other hand, takes this concept even further by enabling models to make predictions on entirely new classes or tasks that weren't seen during training. This is achieved by leveraging semantic information or attributes associated with nodes or edges in the graph.

Transferring knowledge to new graph tasks

Techniques are being developed to apply knowledge from one graph domain to another, enabling rapid adaptation to new problems. Graph meta-learning frameworks, such as **Meta-GNN**, allow models to quickly adapt to new tasks by learning a meta-model that can be fine-tuned with minimal data. This is particularly useful in domains where labeled data is scarce or expensive to obtain.

Meta-learning approaches for graphs

Researchers are exploring meta-learning frameworks that can quickly adapt to new graph tasks with minimal fine-tuning. Approaches such as **graph few-shot learning** (**GFL**) aim to learn transferable knowledge across different graph datasets and tasks. These methods often involve learning a base model that can be quickly adapted to new tasks with just a few examples.

Handling limited labeled data in graph domains

New semi-supervised and self-supervised learning techniques are being created to leverage large amounts of unlabeled graph data effectively. Self-supervised methods such as **graph contrastive learning** (**GraphCL**) and **Deep Graph Infomax** (**DGI**) learn useful representations from unlabeled graph data, which can then be fine-tuned for specific tasks with limited labeled data.

Reinforcement learning on graphs

Combining graph learning with **reinforcement learning** (**RL**) is opening up new application areas.

Graph-based RL for decision-making

Researchers are developing RL algorithms that can operate directly on graph-structured state spaces, enabling more efficient decision-making in complex environments. **Graph convolutional reinforcement learning** (**GCRL**) (`https://arxiv.org/abs/1810.09202`) combines GNNs with RL to handle environments with graph-structured state representations. This approach has shown promise in tasks such as traffic signal control and resource allocation in complex networks.

Applications in robotics and autonomous systems

Graph-based RL is being applied to tasks such as robot navigation and multi-agent coordination, where understanding spatial relationships is crucial. For example, GNNs for decentralized multi-robot path planning have been developed to coordinate multiple robots in complex environments. These approaches allow robots to reason about their environment and other agents more effectively.

Combining GNNs with RL algorithms

Novel architectures that integrate GNNs into RL frameworks are being explored, allowing for more effective learning in graph-structured environments. Approaches such as **relational reinforcement learning** (**RRL**) use GNNs to model relationships between entities in an environment, enabling more sample-efficient learning in complex, structured domains. This has potential applications in areas such as strategic game-playing and complex system optimization.

Integration with other AI domains

The integration of different AI domains has emerged as a key strategy for tackling complex problems and enhancing system performance. This synergistic approach is particularly evident in the integration of graph learning with **large language models** (**LLMs**), federated learning, and quantum computing techniques.

Graph learning and LLMs

The synergy between graph learning and LLMs is, as we learned in *Chapter 6*, a rapidly growing area. Let's explore the future of this relationship.

Enhancing LLMs with graph-structured knowledge

Researchers are exploring ways to incorporate graph-structured knowledge into LLMs, improving their reasoning capabilities and factual accuracy. One approach is to use knowledge graphs as external memory for LLMs, allowing them to access structured information during inference. For example, the **knowledge graph language model** (**KGLM**) integrates a knowledge graph with an LLM, enabling more accurate and contextually relevant text generation.

Another direction is to pre-train LLMs on graph-structured data alongside text. Models such as **Enhanced Language Representation with Informative Entities** (**ERNIE**) incorporate entity embeddings from knowledge graphs during pre-training, resulting in improved performance on entity-related tasks.

Graph-based reasoning in language models

New techniques are being developed to enable LLMs to perform explicit reasoning over graph-structured knowledge, enhancing their ability to answer complex queries. Graph-to-text models such as **Graph2Seq** can generate natural language descriptions of graph structures, bridging the gap between structured knowledge and human-readable text.

Researchers are also developing methods for multi-hop reasoning over knowledge graphs using LLMs. For instance, the **Graph REASoning Enhanced Language Model** (**GREASELM**) framework (`https://arxiv.org/abs/2201.08860`) allows language models to perform step-by-step reasoning over knowledge graphs, improving performance on complex question-answering tasks.

Knowledge graph completion and updating using LLMs

LLMs are being used to automatically expand and update knowledge graphs, creating a symbiotic relationship between textual and graph-based knowledge. Models such as **GPT-3** have shown the ability to generate plausible facts that can be used to populate knowledge graphs. However, ensuring the accuracy of these generated facts remains a challenge.

Techniques such as **zero-shot relation extraction** are being developed to automatically identify new relationships in text and add them to existing knowledge graphs. This allows us to continuously update knowledge graphs based on the latest information that's been extracted from text-by-language models.

Federated graph learning

Federated graph learning (**FGL**) is an innovative approach that combines federated learning principles with graph-based data structures and algorithms. It enables multiple participants to train machine learning models on distributed graph data collaboratively without directly sharing sensitive information. FGL addresses privacy concerns in scenarios involving interconnected data, such as social networks or financial systems. This method allows graph-structured data to be analyzed while maintaining data locality and privacy, making it particularly valuable in domains where data sensitivity and regulatory compliance are crucial.

Distributed learning on decentralized graph data

Federated learning approaches are being adapted for graph data, allowing multiple parties to train models collaboratively without sharing raw data. **Federated graph neural networks** (**FedGNNs**) allow GNNs to be trained across multiple decentralized graph datasets, with only model updates being shared between parties.

Techniques such as vertical federated learning are being explored for scenarios where different features of the same entities are distributed across multiple parties. This allows for collaborative learning on graph data, even when different aspects of the graph are held by different organizations.

Privacy-preserving graph analytics

New techniques are being developed to perform graph analysis while protecting sensitive information, which is crucial for applications in healthcare and finance. Differential privacy methods for graphs, such as edge differential privacy, allow graph statistics to be released while formal privacy guarantees are provided.

Secure **multi-party computation** (**MPC**) techniques are being adapted for graph data, enabling multiple parties to jointly analyze their graph data without revealing sensitive information to each other. This is particularly useful in scenarios where different organizations want to collaborate on graph analysis without sharing raw data.

Applications in healthcare and finance

FGL is enabling collaborative research and model development in sensitive domains such as healthcare and finance, where data privacy is paramount. In healthcare, FGL can be used to analyze patient interaction networks across multiple hospitals without sharing sensitive patient data. This can lead to improved disease prediction and treatment recommendation models.

Expanding on what we discussed in *Chapter 11*, in finance, FGL can be applied to tasks such as fraud detection in transaction networks, allowing multiple financial institutions to collaborate on model development without having to share confidential customer data.

Quantum GNNs

The intersection of quantum computing and graph learning is an exciting frontier.

Leveraging quantum computing for graph problems

Researchers are exploring how quantum algorithms can solve certain graph problems exponentially faster than classical algorithms. **Quantum approximate optimization algorithms** (**QAOA**) have shown promise for solving combinatorial optimization problems on graphs, such as the **maximum cut problem**.

The maximum cut problem is a fundamental graph optimization task where the goal is to partition the vertices of a graph into two sets such that the number of edges between the sets is maximized. By leveraging quantum superposition and interference, QAOA can explore multiple graph partitions simultaneously, potentially leading to faster convergence to near-optimal solutions.

In addition, quantum walk-based algorithms are being developed for tasks such as graph isomorphism testing and centrality computation, potentially offering significant speedups over classical methods for certain graph structures.

Quantum-inspired classical algorithms for graphs

Insights from quantum computing are inspiring new classical algorithms for graph problems, potentially offering significant speedups. Tensor network methods, inspired by quantum many-body physics, are being applied to graph problems such as community detection and link prediction (see *Chapter 7*).

Additionally, quantum-inspired sampling techniques, such as those based on **quantum annealing**, are being adapted for classical computers to solve graph optimization problems more efficiently.

Potential speedups in graph learning tasks

Quantum GNNs are being developed, which could offer dramatic speedups for certain graph learning tasks once quantum hardware matures. Variational quantum circuits are being designed to implement graph convolution operations, potentially allowing for more efficient processing of graph-structured data on quantum hardware.

Hybrid quantum-classical approaches are also being explored, where certain parts of a graph learning pipeline are executed on quantum hardware while others remain classical. This could allow the strengths of both quantum and classical computing paradigms to be leveraged.

Potential breakthroughs and long-term vision

As we explore the frontier of artificial intelligence, it's crucial to consider the potential breakthroughs and long-term vision that could shape the future of this field.

Artificial general intelligence and graphs

Graph representations could play a crucial role in the development of **artificial general intelligence** (**AGI**).

The role of graph representations in AGI

Researchers are exploring how graph-structured knowledge and reasoning could contribute to more general and flexible AI systems. Graph representations offer a natural way to model complex relationships and hierarchies, which is essential for human-like reasoning. For example, the **Neuro-Symbolic Concept Learner** combines GNNs with symbolic reasoning to learn concepts and relationships from visual scenes, demonstrating a potential path toward more general AI systems.

Graph-based world models are being developed to enable AI systems to build and maintain internal representations of their environment. These models can capture causal relationships and allow for planning and reasoning in complex, dynamic environments, which is crucial for AGI.

Graph-based reasoning and common-sense knowledge

Graph representations are being used to capture and reason about common-sense knowledge, a key challenge in AI. Projects such as **ConceptNet** and **ATOMIC** are building large-scale knowledge graphs that encode common-sense facts and relationships. These graphs can be integrated with neural models to enhance their reasoning capabilities and ground their understanding in real-world knowledge.

Researchers are also developing graph-based inference engines that can perform multi-hop reasoning over common-sense knowledge graphs. This allows AI systems to make logical deductions and answer complex queries that require multiple pieces of information to be combined.

Integrating symbolic and neural approaches through graphs

Graphs are serving as a bridge between symbolic AI and neural networks, potentially leading to more powerful hybrid systems. Neural-symbolic integration approaches, such as **logic tensor networks**, use graph structures to combine the strengths of neural networks (learning from data) with symbolic logic (explicit reasoning).

Graph-based neuro-symbolic architectures are being explored for tasks such as visual question-answering and natural language understanding. These systems can leverage both the pattern recognition capabilities of neural networks and the explicit reasoning capabilities of symbolic systems, potentially leading to more robust and interpretable AI.

Neuromorphic computing with graphs

Brain-inspired computing architectures are being explored for graph processing.

Brain-inspired graph architectures

Researchers are developing neural network architectures that more closely mimic the brain's graph-like structure. **Spiking neural networks** (**SNNs**) are being adapted for graph processing tasks, offering potential advantages in terms of energy efficiency and biological plausibility. These models can process information in a more event-driven manner, similar to how neurons in the brain communicate.

Reservoir computing approaches, inspired by the dynamics of biological neural networks, are being applied to graph learning tasks. These models can process temporal graph data efficiently and have shown promise in tasks such as predicting the evolution of dynamic graphs.

Hardware acceleration for graph learning

Specialized hardware is being designed to accelerate graph learning algorithms, potentially leading to dramatic speedups. Neuromorphic chips, such as Intel's Loihi, are being developed to process graph-structured data efficiently and run SNNs. These chips can potentially offer orders of magnitude of improvements in energy efficiency for certain graph learning tasks.

GPUs are being designed specifically for graph computations. These specialized processors aim to overcome the memory bandwidth limitations of traditional architectures when dealing with large, sparse graphs.

Energy-efficient graph processing

New approaches are being explored to make graph learning more energy-efficient and are inspired by the brain's low power consumption. Approximate computing techniques are being applied to graph algorithms, trading off some accuracy for significant gains in energy efficiency. This is particularly important for edge computing applications where power consumption is a critical constraint.

Researchers are also exploring analog computing approaches for graph processing, which can potentially offer extreme energy efficiency for certain graph operations. These include using **memristive devices** to implement GNN operations directly in hardware. Memristive devices are electronic components that can remember their previous resistance state even when power is removed, mimicking the behavior of biological synapses. These nanoscale devices hold great promise for advancing neuromorphic computing, enabling more efficient and brain-like artificial intelligence systems.

Graph learning in the Metaverse

As virtual worlds become more prevalent, graph learning will play a crucial role.

Representing and learning from virtual world graphs

Techniques are being developed to model and analyze the complex networks of interactions in virtual environments. Graph-based representations of virtual worlds can capture spatial relationships, object interactions, and user behaviors. These graph models can be used for tasks such as efficient rendering, physics simulations, and intelligent non-player character behaviors.

Researchers are exploring ways to learn about and update the graph representations of virtual environments in real time, allowing for dynamic and responsive virtual worlds. This includes techniques for online graph learning and incremental graph updates.

Graph-based social interactions in digital spaces

Graph learning is being applied to understand and facilitate social dynamics in virtual worlds. Social network analysis techniques are being adapted for virtual environments to study user interactions, community formation, and information spread. This can help in designing more engaging and socially rich virtual experiences.

Graph-based recommendation systems are being developed for virtual worlds, suggesting connections, activities, or content based on users' interaction patterns and preferences within the virtual environment.

Augmented reality applications of graph learning

Graph-based models are being used to understand and enhance the physical world in **augmented reality** (**AR**) applications. Scene graphs are being employed to represent the spatial and semantic relationships between objects in the real world, enabling more sophisticated AR experiences. GNNs can be used to reason about these scene graphs and predict how virtual objects should interact with the real environment.

Researchers are also exploring graph-based approaches for **simultaneous localization and mapping** (**SLAM**) in AR, using graph optimization techniques to improve the accuracy of spatial mapping and tracking.

Interdisciplinary applications

As we'll see in the following sections, graph learning is finding applications in diverse fields.

Graph learning in climate science and sustainability

Graph-based models are being used to understand complex climate systems and optimize resource allocation for sustainability. Climate networks, where *nodes* represent geographical locations and *edges* represent climate interactions, are being analyzed using graph learning techniques to study phenomena such as El Niño and predict extreme weather events.

In sustainability, graph learning is being applied to optimize smart grids, manage water resources, and design more efficient transportation networks. For example, GNNs are being used to predict energy demand and optimize the distribution of renewable energy sources.

Applications in social sciences and humanities

Researchers are applying graph learning to analyze social networks, study historical texts, and understand cultural phenomena. In sociology, graph learning techniques are being used to study the spread of information and behaviors through social networks. This has applications in understanding phenomena such as the spread of fake news or the adoption of new technologies.

In digital humanities, graph learning is being applied to analyze large corpora of historical texts, uncovering relationships between concepts, authors, and historical events. This can lead to new insights in fields such as literary analysis and historical research.

Graph-based approaches in economics and finance

Graph learning is being used to model economic networks, predict market trends, and detect financial fraud. In economics, researchers are using GNNs to model supply chain networks and predict the impact of disruptions. This has become particularly relevant in light of recent global supply chain challenges.

In finance, graph-based anomaly detection techniques are being developed to identify complex fraud patterns in transaction networks. Graph learning is also being applied to analyze financial markets, model relationships between different financial instruments, and predict market movements.

These interdisciplinary applications demonstrate the broad potential of graph learning to tackle complex problems across various domains, potentially leading to significant breakthroughs in how we understand and manage complex systems.

Summary

In this chapter, we explored the exciting future of graph learning, highlighting key trends and advancements shaping this dynamic field. We discussed upcoming directions in scalability and efficiency, focusing on techniques that you can use to handle larger and more complex graphs, distributed learning algorithms, and graph compression methods. We delved into the growing importance of interpretability and explainability in graph models, as well as advancements in handling dynamic and temporal graphs. We also covered the challenges and opportunities presented by heterogeneous and multi-modal graphs and explored advanced architectures such as graph transformers and generative models.

Then, we examined the integration of graph learning with other AI domains, such as LLMs and RL, along with privacy-preserving techniques in FGL. In addition, we touched on the potential of quantum GNNs and the role of graph learning in AGI. Finally, we discussed interdisciplinary applications of graph learning in fields such as climate science, social sciences, and economics, showcasing the broad impact and potential of this technology across various domains.

As this is our final chapter, we want to remind you of all the skills you've acquired throughout this book and how you can apply them, from understanding why we need graphs to using graph deep learning for real-world applications. To continue your journey, we recommend exploring further research in specialized areas that interest you most, whether that involves GNNs, knowledge graphs, or graph-based recommender systems. You may also consider participating in graph learning competitions or contributing to open-source graph learning projects to gain practical experience.

Remember, the field of graph learning is constantly evolving, so staying updated with the latest research papers and attending relevant conferences or workshops will help you remain at the forefront of this exciting domain.

Index

Other Books You May Enjoy