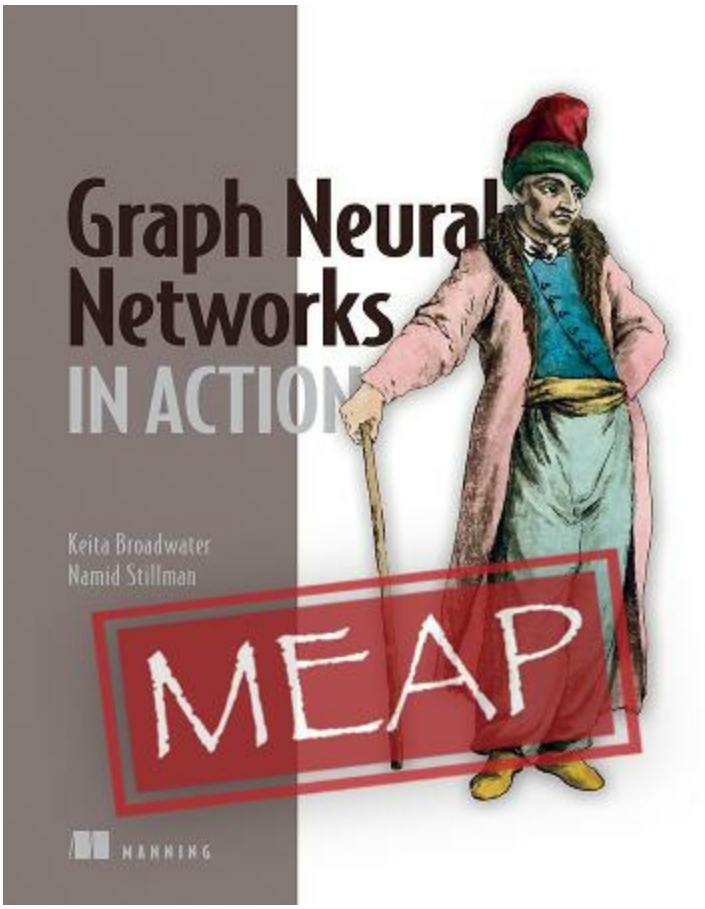
Graph Neural Networks IN ACTION

Keita Broadwater
Namid Stillman

MEAP

MANNING


Graph Neural Networks
IN ACTION
Keita Broadwater
Namid Stillman
MEAP
MANNING

Graph Neural Networks in Action MEAP V06

1. Copyright_2023_Manning_Publications
2. welcome
3. 1_Discovering_Graph_Neural_Networks
4. 2_System_Design_and_Data_Pipelining
5. 3_Graph_Embeddings
6. 4_Graph_Convolutional_Networks_(GCNs)_&_GraphSage
7. 5_Graph_Attention_Networks_(GATs)
8. 6_Graph_Autoencoders
9. Appendix_A._Discovering_Graphs

MEAP Edition

Manning Early Access Program

Graph Neural Networks in Action

Version 6

For more information on this and other Manning titles go to

manning.com

welcome

Hello,

Thanks for purchasing *Graph Neural Networks in Action*. If you want a resource to understand and implement GNNs, this book is for you! When I was dipping my toes into GNNs, most resources were tutorials on Medium.com, GNN library documentation, some papers with code, and a few scattered videos on youtube. These were useful sources of knowledge, but learning from them was inefficient.

These resources were scattered about the internet. And, in many of them, there were unspoken assumptions and prerequisites that presented roadblocks to comprehensive understanding. Finally, many tutorials and explanations lacked the nuance to allow me to answer the questions of discernment like, ‘why use a GCN over GraphSage ?’.

I wrote GNNs in Action to remedy many of these pain points. I structured it to be both a book where you can jump into a topic of interest and get something down in code, and as a reference that comprehensively covers the relevant knowledge that would allow one to employ and discuss GNNs with confidence. This is a hard balance to strike, and I hope to have done it well.

Speaking of implementation, one other thing that bothers me is when a book uses frameworks and languages that I don’t use. Though I use python exclusively as the programming language, examples can be found that are grounded in the two major frameworks, pytorch and tensorflow. For smaller datasets, I use Pytorch Geometric and DGL; for scaled problems, I use Alibaba’s Graphscope, as well.

If you are an intermediate user of python, and havesome data science or machine learning engineering experience, you should be good enough to dive in. I also assume you have basic knowledge of neural networks and linear algebra.

I hope this book is a valuable resource for you to get your work project into production, to have informed discussions with colleagues, or to make your own toy projects and tutorials. Your frank feedback, comments, questions, and reviews are welcome in the livebook discussion forum, so that this work can be improved.

Happy Reading,

Keita Broadwater

In this book

Copyright 2023 Manning Publications welcome brief contents 1 Discovering Graph Neural Networks 2 System Design and Data Pipelining 3 Graph Embeddings 4 Graph Convolutional Networks (GCNs) & GraphSage 5 Graph Attention Networks (GATs) 6 Graph Autoencoders
Appendix A. Discovering Graphs

1 Discovering Graph Neural Networks

This chapter covers

- Defining Graphs and Graph Neural Networks (GNNs)
- Understanding why people are excited about GNNs
- Recognising when to use GNN
- A big picture look at solving a problem with a GNN

“Sire, there is no royal road to geometry.”

Euclid

For data practitioners, the fields of machine learning and data science initially excite us because of the potential to draw non-intuitive and useful insights from data. In particular, the insights from machine learning and deep learning promise to enhance our understanding of the world. For the working engineer, these tools promise to deliver business value in unprecedented ways.

Experience detracts from this ideal. Real data is messy, dirty, biased. Statistical methods and learning systems have limitations. An essential part of the practitioner’s job involves understanding these limitations, and bridging this gap to obtain a solution.

For a certain class of data, graphs, the gap has proven difficult to bridge. Graphs are a data type that is rich with information. Yet, they can also explode in size when we try to account for all this information. They are also ubiquitous, appearing in nature (molecules), society (social networks), technology (the internet), and everyday settings (roadmaps). In order to use this rich and ubiquitous data type for machine learning, we need a specialized form of neural network dedicated to work on graph data: this is the graph neural network or GNN.

Some basic definitions and an example will help us get a better idea of what GNNs are, what they can do, and where they can be helpful. We'll be reviewing the basics of graphs and how problems that use tabular data can be recast as graph–based problems, using this to predict survivorship from the Titanic dataset or how drug discovery is being aided by understanding molecules as graphs. We hope that the rest of this chapter helps to make you as excited as us about the power of graph-based learning and we look forward to demonstrating to you the world of graph neural networks in action.

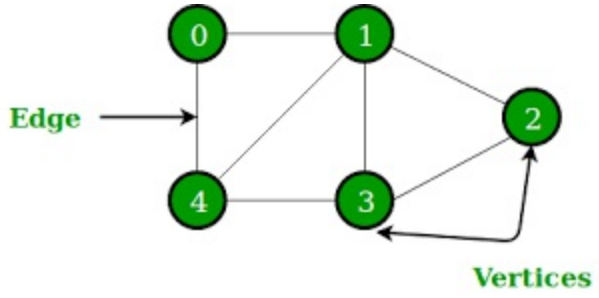
1.1 Graph-based Learning

What are graphs?

Before we can understand how to make learning algorithms that work on graphs, we need to let's define graphs more precisely.

Graphs are data structures with elements, expressed as nodes or vertices, and relationships between elements, expressed as edges. The links are key to the power of relational-data, as they allow us to learn more about the system, give new tools for analyzing data, and predict new properties from it. This is in contrast to tabular data that is common in traditional learning and analytics problems, such as a database table, dataframe, or spreadsheet.

Figure 1.1 A graph. Individual elements, represented here by numbers zero to four, are nodes, also called vertices, and their relationships are described by edges, also known as links.



In order to describe and learn from the edges between the nodes, we need a way to write them down. This could be done explicitly, stating that the zero node is connected to one and four, and that the one node is connected to zero, two, three and four. Quickly, we can see that this becomes unwieldy and that

we’re repeating redundant information (that zero is connected to one and that one is connected to zero). Luckily, there are many mathematical formalisms for describing relations in graphs. One of the most common is to describe the *adjacency matrix*, which we write out below.

Figure 1.2 The adjacency matrix for the graph in Figure 1.1. Notice that the adjacency matrix is symmetric across the diagonal and that all values are ones or zero. This means the graph is undirected and unweighted.

	0	**1**	**2**	**3**	**4**
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

An adjacency matrix makes it easy to write out all the different connections in a single table. Here, we assumed that all edges point in both directions, namely that if zero is connected to one then one is also connected to zero. This is known as an *undirected graph*. Undirected graphs can be easily worked out from an adjacency matrix because the matrix is symmetric across the diagonal (the upper right triangle is reflected onto the bottom left). We also assume here that all the relations are the same. Instead, if we wanted the relation of one to four to mean more than the relation of one to zero then we could increase the weight of this edge. This translates to increasing the value in the adjacency matrix, making the entry for the four-one edge equal to ten, for example. Graphs where all relations are of equal importance are known as *unweighted graphs* and can also be easily observed from the adjacency matrix because all graph entries are either ones or zero. Finally, we note that

all the nodes in the graph have zeros at their own value in the adjacency matrix (zeros along the diagonal). Here we assume that graphs don't have *self-loops*, where edges connect back to themselves. To introduce this, we just make the value for that node non-zero at their position in the diagonal.

In practice, an adjacency matrix is only one of many ways to describe relations in a graph. Others include adjacency lists, edge lists, or an incidence matrix. Understanding these types of data structures well is vital to graph-based learning. If you're unfamiliar with these then we recommend looking through Appendix A, where we've given additional details for those new to graphs.

Recasting tabular data as a graph-based learning problem

To introduce graph-based learning, we're going to recast a classic machine learning problem in a new light by using graphs. By doing so, we can get an idea of how such problems can be represented as networks, and see possible applications of GNNs.

The Titanic dataset, which describes passengers of the Titanic ship, many of whom famously met an untimely end when colliding with an iceberg. This dataset has historically been described in tabular format, but is full of unexplored relationships. We're going to describe some of these hidden relations and the types of predictions that graph-based learning can help us make. This dataset is commonly used for classification and has a target variable (survivorship) as well as columns of features which fit well into a dataframe, where we provide a snapshot in Figure 1.3. Each row represents one person.

Figure 1.3 The Titanic Dataset is usually displayed and analyzed using a table format.

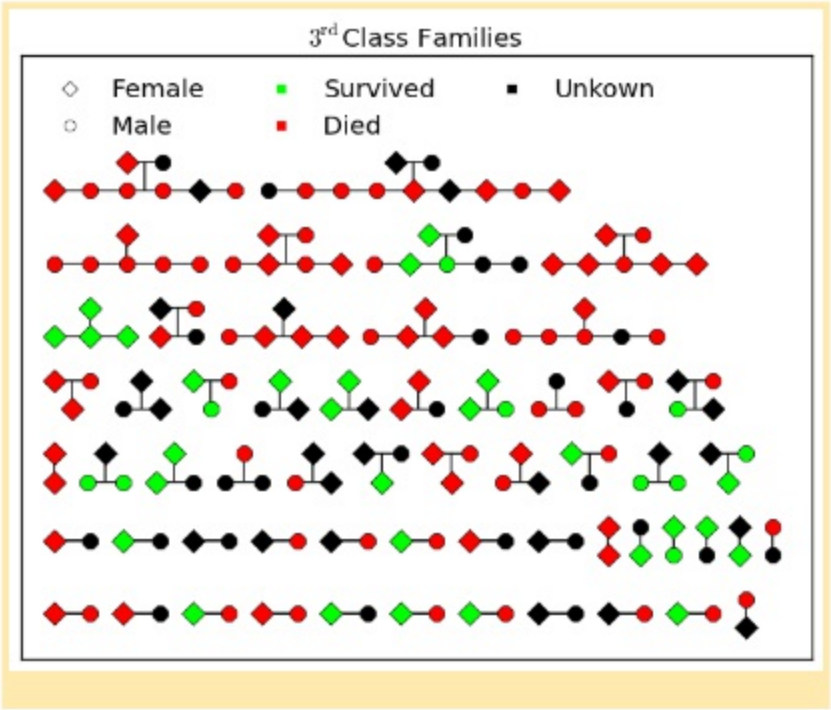
	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

As expressive as tables like this are, the accounts of relationships are typically hidden. For example, it's hard to work out the social links between the people in a dataset like this, or the corridors that link locations on the ship.

We know that many of the people on the ship share multiple familial types of relationships, including marital and family relations (married, engaged, parent/child, siblings, and cousins), as well as business and employment relationships, and, of course, both new and old friendships. Tabular versions of the Titanic dataset give high level data on this such as boolean indicators (e.g., in the table above, the alone feature is false if a person had immediate family on the ship) or counts of immediate family relationships.

Graph representations of social relationships add depth via specificity. For example, having relatives on the ship may be a factor in favor of survival, and having socially important relatives might give an even greater chance at survival. Social networks can convey social importance via their structure. An example of a graph representation of all the families on the Titanic is shown in figure 1.4. What's interesting in this graph is that there are many passengers with unknown family ties. As we shall discuss, graph-based learning allows us to learn these data as a task known as *node prediction*.

Figure 1.4 The Titanic Dataset. The family relationships of the Titanic visualized as a graph (credit: Matt Hagy). Here we can see that there was a rich social network as well as many passengers with unknown family ties.

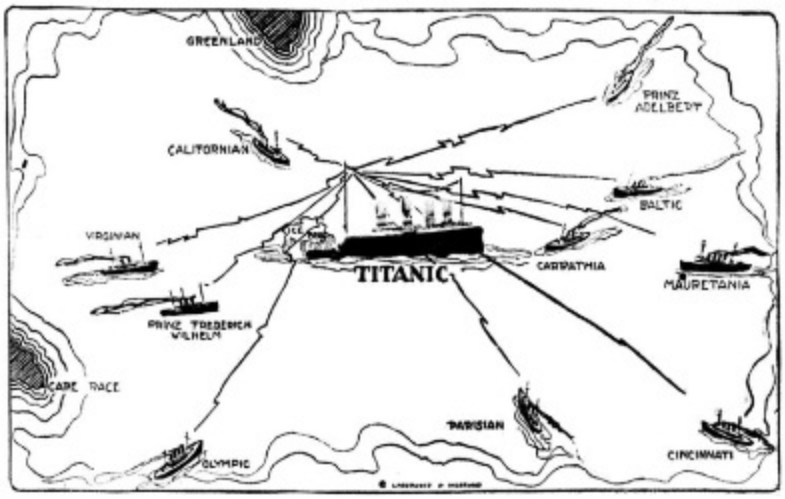


It’s not just the passengers on the Titanic that can be understood as a graph but also the Titanic itself. The original boat, now lost to the sea, was massive, spanning more than 200 metres and containing ten decks (eight of which were for passengers). The ship’s corridors, consisting of connected hallways, stairways, junctions and landings, and the adjoining cabins and workrooms can all be represented as a graph. These are important to the survival question because the ease to quickly get to a lifeboat in a crisis depends on one’s starting location. From certain cabins and decks, it would be easier to reach a lifeboat in time than others. When working out an escape route, graph-based learning can help us again. Tasks where we predict edges (such as whether this edge should be part of the escape route) are known as *edge-prediction tasks*.

Finally, the Titanic also existed within a wider network of other ships and maritime vessels. These ships communicated via wireless telegraph and by analog signals, such as flares, and this allowed the ship’s crew to communicate between themselves, to the passengers, and to the outside world (Figure 1.5). The communication network, or graph, is also relevant to the question of survival as critical information about the crisis could only reach individuals within close-proximity to the communication nodes. The

communication networks of the Titanic and surrounding ships, impacted survivability. There were two ships within 14 miles of the Titanic when it sank. They saw the distress flare, but for different reasons, decided not to help. Again, graph-based learning could have helped in a modern-day rescue mission of the Titanic. Working out the optimal communication network, for survivability, which would also be a graph, would be achieved using tasks known as *graph-prediction.*

Figure 1.5 An old illustration of the wireless communication between the Titanic and other ships. It's easy to interpret the ships as nodes and the wireless connections as edges.



We have seen how many problems in the real world are actually graph-based problems in disguise. These can be understood as *node prediction*, *link prediction* and *graph prediction tasks*. We'll give more details and see other examples of these tasks later in this chapter but for those wanting to skip ahead, we deal with node prediction in Chapter 5, link prediction in Chapter 6 and graph prediction in Chapter 7. Each of these also describe specific architectures of graph neural networks, a deep learning method that is specifically designed to work on graph-based or relational data.

What are graph neural networks?

A **graph neural network (GNN)** is an algorithm that allows you to represent and learn from graphs, including their constituent nodes, edges and features. Graph neural networks provide analogous advantages to conventional neural

networks, except the learning happens on relational data. Applying traditional machine learning and deep learning methods to graph data structures has proven to be challenging. This is because graph data, when represented in grid-like formats and data structures, can lead to massive repetitions of data. In particular, graph-based learning focuses on approaches that are *permutation invariant*. This means that the machine learning method is uninfluenced by the ordering of the graph representation. In concrete terms, it means that if we shuffle the rows and columns of the adjacency matrix then our algorithms performance shouldn't change. While this might seem obvious, we can imagine that this doesn't hold true for images. If we shuffle the rows and columns of an image then we scramble the input. Instead, machine learning algorithms for images seek *translational invariance*, which means that we can translate (shift) the object in an image and it won't impact the performance of the algorithm. These invariances, also known as inductive biases, are powerful tools for designing machine learning algorithms and permutation invariant approaches is one of the central reasons that graph-based learning has increased in popularity in recent years. GNNs are the poster-child for permutation invariance and we'll be exploring them in more depth throughout this book.

However, the concept of applying specialized neural network architectures to graphs is a relatively old idea, being first introduced in the late nineties. It gained traction in the early 2000s, and has greatly accelerated after 2015 ([Wu] has a good list of these papers). Today, there are many examples of enterprise systems that employ GNNs at scale, including Pinterest, Amazon, Uber, and Google. However, this is also a rapidly developing field, with many challenges still remaining. This book will give you the skills and resources to implement the right GNN for your own problems, while also giving some indication of where GNNs aren't appropriate.

1.2 Goals of the book

GNNs in Action is a book aimed at practitioners who want to deploy graph neural networks to solve real problems. This could be a machine learning engineer not familiar with graph data structures, or a data scientist who has not tried deep learning, or software engineers who may not be familiar with either.

Our aim throughout this book is to enable you to

- Assess the suitability of a GNN solution for your problem.
- Implement a GNN architecture to solve problems specific to you.
- Understand the tradeoffs in using the different GNN architectures and graph tools in a production environment.
- Make clear the limitations of GNNs.

As a result, this book is weighted towards implementation using programming and mainstream data ecosystems. We devote just enough space on essential theory and concepts, so that the techniques covered can be sufficiently understood. We also should mention that GNNs are a relatively new tool in the machine learning library and extensions are constantly being suggested. As such, we will only be describing some of the most popular architectures and note that there are many many more to explore.

We'll be covering the end-to-end workflow of machine learning,including data loading and preprocessing (Chapter 2), model setup and training (Chapter 3), and inference (Chapters 4 to 7). We'll also touch on how we can scale these algorithms (Chapter 8) which is a common challenge in deploying these methods. Finally, we'll be reviewing practical guidelines in implementing machine learning solutions, including prototyping and developing applications that will be put into production, and especially as relevant to GNNs.

GNNs in Action is a book designed for people to jump quickly into this new field and start building applications. Our aim for this book is to reduce the friction of implementing new technologies by filling in the gaps and answering key development questions whose answers may not be easy to find, or not covered at all.

This book is set out as follows; in Part 1, we cover the fundamentals of machine learning workflows and data pipelines, especially for graph-based learning. We devote Chapter 2 to a tour of graphs and the graph technical ecosystem. Chapter 3 covers the graph data pipeline from raw data to graph representations in more detail. For readers familiar with graphs, this material may seem elementary. For those readers, we suggest skimming these chapters and reviewing topics as needed, then jumping on to chapters on Graph Neural

Networks, which start in Chapter 3.

In Part 2, the core of the book, you'll get introduced to the major types of GNNs, including Convolutional Networks (GCNs) and GraphSage, Graph Attention Networks (GATs), and Graph Auto-Encoders (GAEs). These methods are the bread and butter for most GNN applications and also cover a range of other deep learning concepts such as convolution, attention, and autoencoders.

Finally, in Part 3, we'll look at more advanced topics. We describe GNN for temporal data (Dynamic GNNs) in Chapter 7 and give methods for GNNs at scale in Chapter 8. While these chapters are meant more for the reader that is familiar with the earlier material in the book, we note that these methods are frequently used in practice.

Throughout our book, python is the coding language of choice. There are now several GNN libraries that exist in the python ecosystem, including Pytorch Geometric, Deep Graph Library, GraphScope, and Jiraph. We focus on PyTorch Geometric (PyG) which is one of the most popular and easy to use frameworks. We want this book to be approachable by an audience with a wide set of hardware constraints, and try to reflect this in the code examples. Hence, code snippets throughout are available on Github and in Colab notebooks.

1.3 Why are people excited about GNNs?

Deep learning is often thought of as a field filled with hype around new technologies and methods. However, GNNs now widely recognised as a genuine leap forward for graph-based learning. This doesn't mean that GNNs are a silver bullet and we strongly advise to still compare GNN-based methods to simple benchmarks such as linear regression or XGBoost. This is especially true where the data is in tabular form. However, where you expect there to be a strong relational component in your data science problem, GNNs might just be the missing part of the puzzle that you've been looking for all along.

The recent that GNNs are so helpful is because they put relational

dependencies front and center in the learning process. Previously if one wanted to incorporate relational features from a graph into a deep learning model, it had to be done in an indirect way. These often couldn't be scaled, and could not seamlessly take into account node and edge properties.

That’s not to say that those working with graphs were completely lost for methods to use. The field of graph analytics and graph-based learning is an old and rich field that contains many many methods to characterize graphs. Many of these methods are in fact combined with GNN-based approaches and are still commonly used in modern graph databases and analytical software [Deo]. However, such methods can fall short when applying them to very large data, especially where we want to have methods that can generalize to many graphs of different shapes and sizes. In effect, GNN-based methods are leading the field in conventional graph-based learning benchmarks. Because of this, they are a vital tool to be aware of when working with relational data.

Below, we’ll give additional examples for some of the applications of GNNs that we have found most exciting. These are, of course, just the tip of the iceberg. In fact, GNNs can be broadly grouped into two categories of applications,

1. Applications based on new and unique data segments and domains; and
2. Applications that enhance traditional machine learning problems in fresh ways.

The following examples include two novel applications of GNNs (category 1), and one example of using a GNN in a traditional application (category 2).

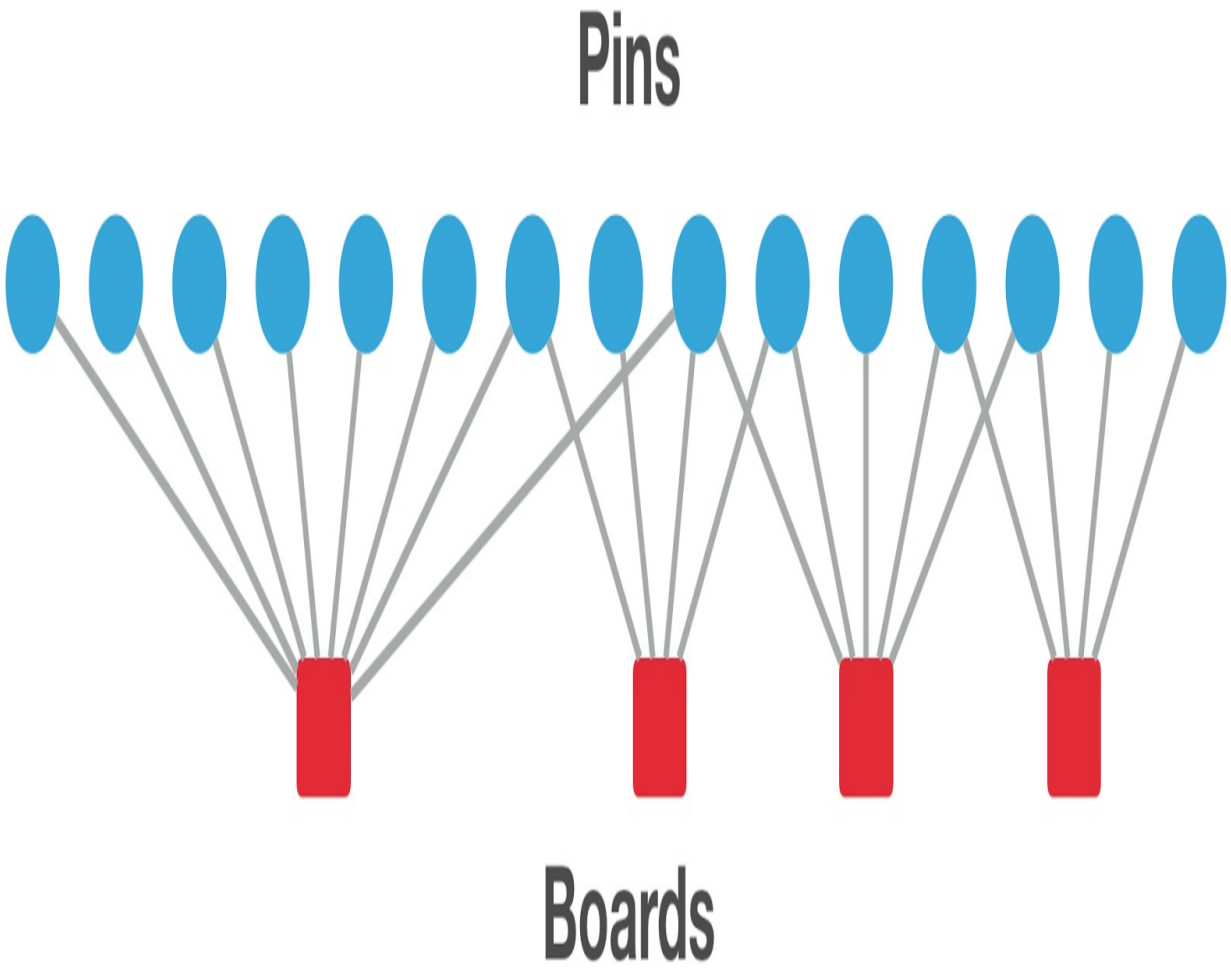
Recommendation Engines

Many of the applications in graph-based learning are driven by scientific discovery such as drug design and inferring physical laws. We’ll give a brief overview of both of these problems below. However, we understand that many practitioners want to know how to apply graph-based learning to enterprise problems and at scale.

Enterprise graphs can exceed billions of nodes and many billions of edges. On the other hand, many GNNs are benchmarked on datasets that consist of fewer than a million nodes. When applying GNNs to large graphs, adjustments of the training and inference algorithms, storage techniques, all might have to be made.

One of the most famous examples of GNNs being applied to industry problems at scale have been in constructing recommendation engines. For example, Pinterest is a social media platform for finding and sharing images and ideas. There are two major concepts to Pinterest's users: collections or categories of ideas, called *boards* (like a bulletin board); and objects a user wants to bookmark called *pins*. Pins include images, videos, and website URLs. For example, a user board focused on dogs would include pins of pet photos, puppy videos, and dog-related website links. A board's pins are not exclusive to it; a pet photo that was pinned to the dog board could also be pinned to a 'pet enthusiasts' board (shown in Figure 1.6).

Figure 1.6 A graph representing the 'pins' and 'boards' of Pinterest. Pins make up one set of nodes (circles) and the boards make up the other set (squares). In this particular graph, nodes can only link to nodes of the other group.



As of this writing, Pinterest has 400M active users, who have likely pinned tens if not hundreds of items per user. One imperative of Pinterest is to help their users find content of interest via recommendations. Such recommendations should not only take into account image data and user tags, but draw insights from the relationships between pins and boards.

One way to interpret the relationships between pins and boards is with a special type of graph called a *bipartite graph*. Here, pins and boards are two classes of nodes. Members of these classes can be linked to members of the other class, but not to members of the same class. This graph was reported to have 3 billion nodes and 18 billion edges.

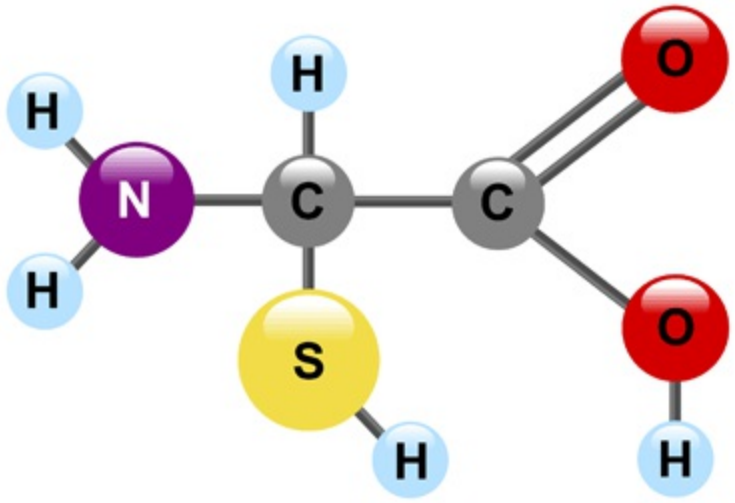
PinSage, a graph convolutional network, was one of the first documented highly scaled GNNs used in an enterprise system. Used at Pinterest to power recommendation systems on its massive bipartite graphs (consisting of object

pins linked to boards), this system was able to overcome challenges of GNN models that weren't optimized for large scale problems. Compared to baseline methods, AB tests on this system showed it improved user engagement by 30%. This is an example of edge prediction, where we learn to predict which objects should be recommended to be included in a users graph. However, GNNs can also be used to predict what an object is, such as whether it contains a dog or mountain, based on the rest of the nodes in the graph. This task is node prediction. We'll be learning about node prediction tasks as well as graph convolutional networks, of which PinSage is an extension of, in Chapter 4.

Drug discovery

In chemistry and molecular sciences, a prominent problem has been representing molecules in a general, application-agnostic way, and inferring possible interfaces between organic molecules, such as proteins. For molecule representation, we can see that the drawings of molecules that are common in high school chemistry classes bear resemblance to a graph structure, consisting of nodes (atoms) and edges (atomic bonds).

Figure 1.7 A molecule. We can see that individual atoms are nodes and the atomic bonds are edges.

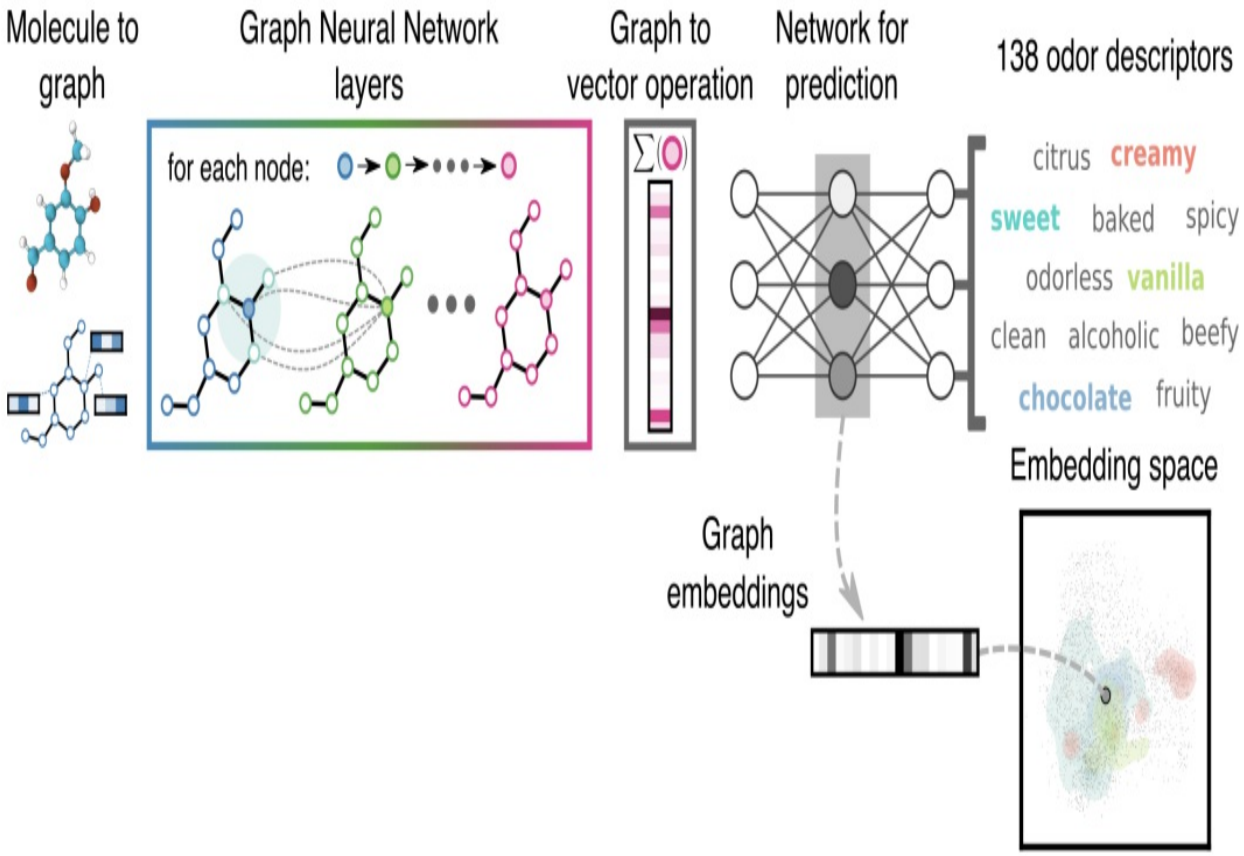


Applying GNNs to these structures can, in certain circumstances, outperform traditional 'fingerprint' methods. These traditional methods involve the

creation of features by domain experts to capture a molecule's properties. This is extremely important for predicting whether a chemical is toxic or safe for use or whether it has some downstream effects which can impact disease progression. Hence, GNNs have shown to be incredibly useful for the field of drug discovery. For more information, we refer the interested reader to **Molecular Fingerprints and Interfaces [Sanchez-Lengeling].**

Drug discovery, especially for GNNS, can be understood as a *graph prediction* problem. We learnt earlier in this chapter when discussing the communication network of the Titanic, that graph prediction tasks are those that require learning and predicting properties about the entire graph. For drug discovery, the aim is to either predict properties such as toxicity or treatment effectiveness of small molecular graphs (classification) or to suggest entirely new graphs that should be synthesized and tested (regression). To suggest these new graphs, drug discovery methods often combine GNNs with generative models such as variational autoencoder, as shown for example in Figure 1.8. We'll be describing variational graph autoencoders (VGAE) in detail in Chapter 6.

Figure 1.8 A GNN system used to match molecules to odors. The workflow here starts on the left with a representation of a molecule as a graph. In the middle parts of the figure, this graph representation is transformed via GNN and MLP layers to a vector representation that can be trained against odor descriptions on the right. Finally a graph representation of the molecule can be cast in a space which also corresponds to odors (bottom right).



Mechanical Reasoning

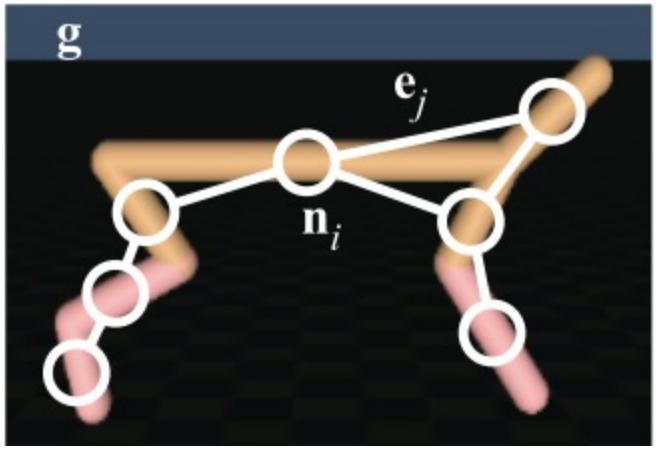
We develop rudimentary intuition about mechanics and physics of the world around us at a remarkably young age and without any formal training in the subject. Given a bouncing ball, we don't need pencil and paper and a set of equations to predict where it might fall and catch it. We don't even have to be in the presence of a physical ball. Given a series of snapshot of a bouncing ball, we can reasonably predict in a 3D world where the ball might end up.

While these problems might seem trivial, they are critical for many industries such as manufacturing and autonomous driving. For example, autonomous driving systems need to anticipate what will happen in a traffic scene consisting of many moving objects. Until recently, this task was typically treated as a problem of computer vision. However, more recent approaches have begun to use GNNs. These GNN-based methods demonstrate that including relational information can enable algorithms to develop physical intuition with higher accuracy and with fewer data. For more information, we

refer the reader to [Sanchez-Gonzalez].

In Figure 1.9, we give an example of how a body can be thought of as a ‘mechanical’ graph. The input graphs for these physical reasoning systems consist of nodes at central points on the body where, for example, limbs connect, or individual objects, such as bouncing balls. S. The edges of the graph then represent the physical relationship (such as gravitational forces, elastic springs, or rigid connections) between the nodes. Given these inputs, GNNs learn to predict future states of a set of objects without explicitly calling on physical/mechanical laws. Hence, these methods are a form of *edge prediction*, that is they predict how the nodes connect over time. Furthermore, these models have to be dynamic to account for the temporal evolution of the system. We consider these problems in detail in Chapter 7.

Figure 1.9 A graph representation of a mechanical body. The body’s segments are represented as nodes, and the mechanical forces binding them are edges.



1.4 How do GNNs work?

In this chapter, we have seen many different examples of graph-based problems and how these can form the basis for many graph-based learning tasks. Throughout, we have alluded to graph neural networks (GNNs) as a new tool to help solve these and many other graph-based problems. However, we have yet to describe the basics of what exactly GNNs are and how they work. We’re now at the stage of the story where we pull back the curtain and reveal the basics of GNNs.

Though there are a variety of GNN architectures at this point, they all tackle the same problem of dealing with unstructured graph data in a way that is permutation invariant. The mechanism by which they do this is by encoding and exchanging information across the graph structure during the learning process.

In a conventional neural network, we first need to initialize a set of parameters and functions. These include the number of layers, the size of the layers, the learning rate, the loss function, and the batch size. These are all treated in more detail elsewhere and we assume the reader is familiar with these terms. Once we have defined these features, we then train our network by iteratively updating these parameters. Explicitly, we perform the following steps:

A Input our data

B Pass the data through neural network layers that transform the data according to the parameters of the layer and an activation rule.

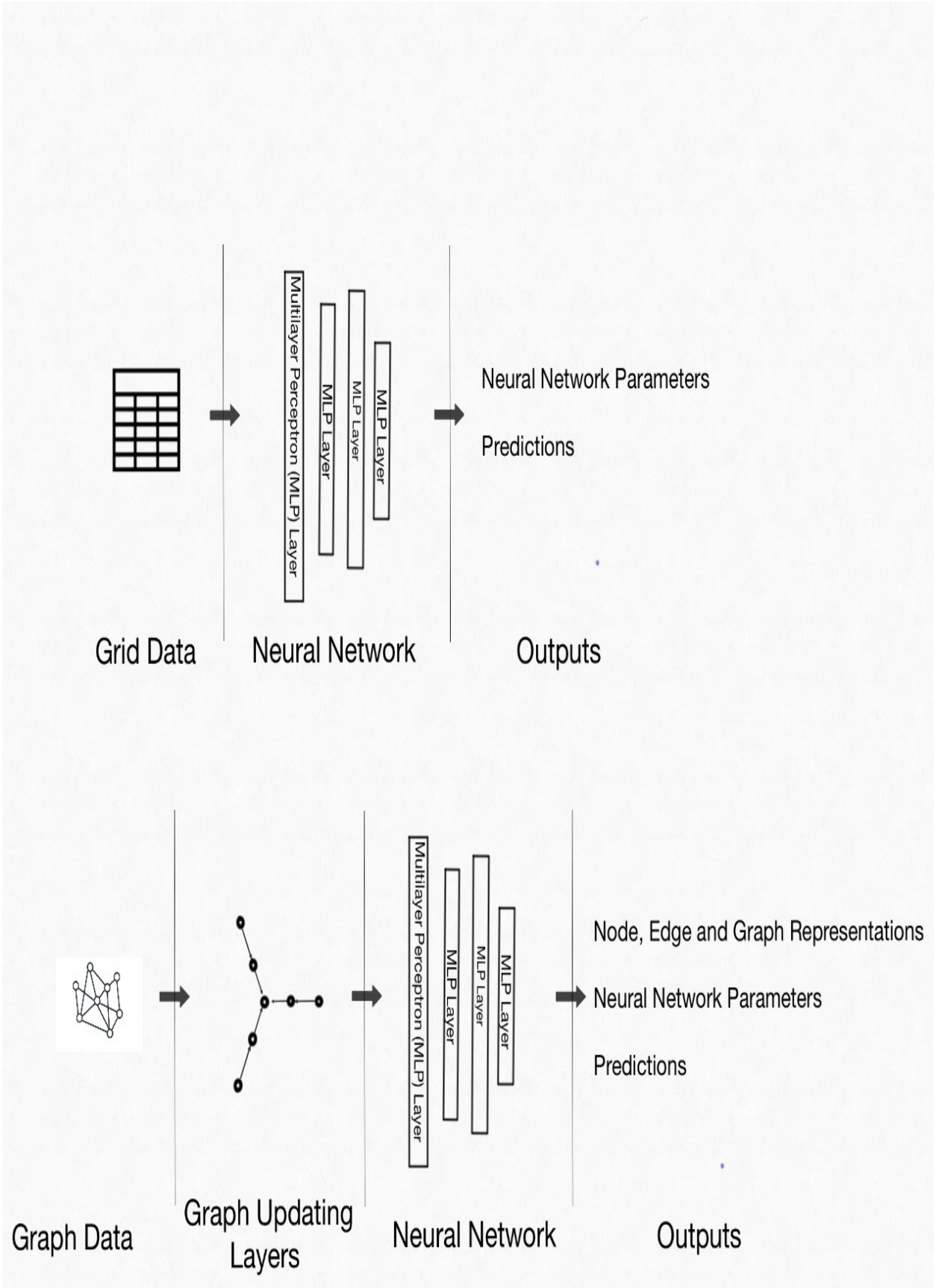
C Output a representation from the final layer of the network.

D Backpropagate the error and adjust the parameters accordingly.

E Repeat these steps a fixed number of epochs or until training has converged.

For tabular data, these steps are exactly as above, as shown in Figure 1.9. For graph-based or relational data, these steps are similar except that each epoch relates to one iteration of *message passing.*

Figure 1.10 Comparison of (simple) non-graph neural network (above) and graph neural network. GNNs have a layer that distributes data amongst its vertices.


Grid Data
Neural Network
Multilayer Perceptron (MLP) Layer
MLP Layer
MLP Layer
MLP Layer
Outputs
Neural Network Parameters
Predictions
Graph Data
Graph Updating Layers
Neural Network
Multilayer Perceptron (MLP) Layer
MLP Layer
MLP Layer
MLP Layer
Outputs
Node, Edge and Graph Representations
Neural Network Parameters
Predictions

1.4.1 Message Passing

One of, if not the, major aspect that separates GNNs from other deep learning methods lies with message passing. Message passing is how GNNs encode information about the graph. This introduces an additional step to our training process. We now have graph-updating layers which allow us to encode and propagate local information which is passed to our downstream machine learning tasks. Hence, we still have to initialize the neural network parameters, but we now also initialize a representation of the nodes of the graph, as shown in Figure 1.9. In our iterative process, we now

A Input our graph data

B Update the node or edge representations using message passing

C Pass the node/edge representations to neural network layers

D Output a representation from the final layer of the network.

E Backpropagate the error and adjust the parameters accordingly.

F Repeat these steps a fixed number of epochs or until training has converged.

Message passing is designed specifically for interrogating the graph structure. For each node in the graph, each message passing step represents a communication that spans nodes one hop away. With each GNN layer, we propagate messages one hop further. Hence, even for large graphs, a handful of hops can cover an exponentially large proportion of the graph. For this reason, GNN layers are almost never as ‘deep’ as a deep learning network can be. From a performance point of view, there is a law of diminishing returns in applying many layers. We’ll get into further specifics of this as we review each architecture.

Most of this book will be focused on the middle sections of the diagrams in Figure 1.8, namely the different ways message passing is done. We study these in detail in Chapters 3 through to 8.

1.5 When to use a GNN?

We have described shared real applications of GNNs: studying molecular fingerprints, deriving physical laws and motion, and improving a recommendation system. In order to work out when to use GNNs, we can consider their shared commonalities. Listing these out can give us a hint as to where a GNN may be a good fit. In particular, we can see that for each one

- Data was modeled as a graph.
- Nodes and edges have non-trivial information
- A prediction task was involved.

Let's flesh out these points below.

1.5.1 Data modeled as a graph

Graphs are a very versatile data structure, so with a little imagination, many problems can be framed in this way. A critical consideration is whether the relationships being modeled are of relevance to the problem at hand. A graph-based model is almost always helpful when relationships are an important characteristic for the situation being modeled.

We've already seen many examples of this. Table 1 shows this explicitly, describing how we can assign nodes and edges to express those relationships.

Table 1.1 Comparison of graph examples discussed in this chapter, particularly showing what nodes and edges represent. We see that nodes and edges can represent a range of concrete and abstract entities and ideas.

Problem	Nodes	Edges
Titanic Dataset	a) Individual People b) Corridor Junctions c) Communication	a) Societal Relationships b) Ship Corridors

	Hubs	c) Communication Links
Molecular Fingerprints	Atoms	Atomic Bonds
Mechanical Reasoning	Physical Objects	Forces and Physical Connections
Pinterest Recommender	Pins and Boards	User-created Pin/Board Links

In contrast, for many applications, individual data entries are meant to stand alone, and relationships are not useful. For example, consider a non-profit organization that has maintained a database of donors and their financial contributions. These donors are spread across the country, and for the most part don't personally know each other. Also, the non-profit is primarily concerned with growing donor contributions. Representing their donor data with a graph data structure and storing it using a graph database would be a waste of time and money. In general, it is wasted effort to try to squeeze tabular data into a graph-based model when relational information isn't important.

1.5.2 Incorporation of node and edge features

We've already described how graph-based models should be built for relational problems. This might seem obvious, but it's always important to keep this in mind. However, once we have relational information, the next step is to get as much information as possible into our graphs.

In many graphs, nodes and edges have properties associated with them that can straightforwardly be used as features in a machine learning problem. For example, in the Pinterest case above, a pin that is the image of a dog may have user-generated tags attached to it (e.g., 'dog', 'pooch'). The node that

represents this pin could thus have features related to the text data and the image.

However, sometimes there is also non obvious information that can also be leveraged for graph-based inference. In the Pinterest case, we could use image segmentation on the images or some other computer vision approach to further decode saved images into nodes. Alternatively, if we're working with graphs of text documents, such as scientific papers, we can use natural language processing (NLP) methods to put more information into our nodes. We'll see this later in the book, in Chapter 4 and 6, where principal component analysis is used to summarise user reviews into 100 features for each node in a consumer purchaser network. In general, if there is information in the data, that information should also appear in some form in the graph.

1.5.3 A prediction task is involved

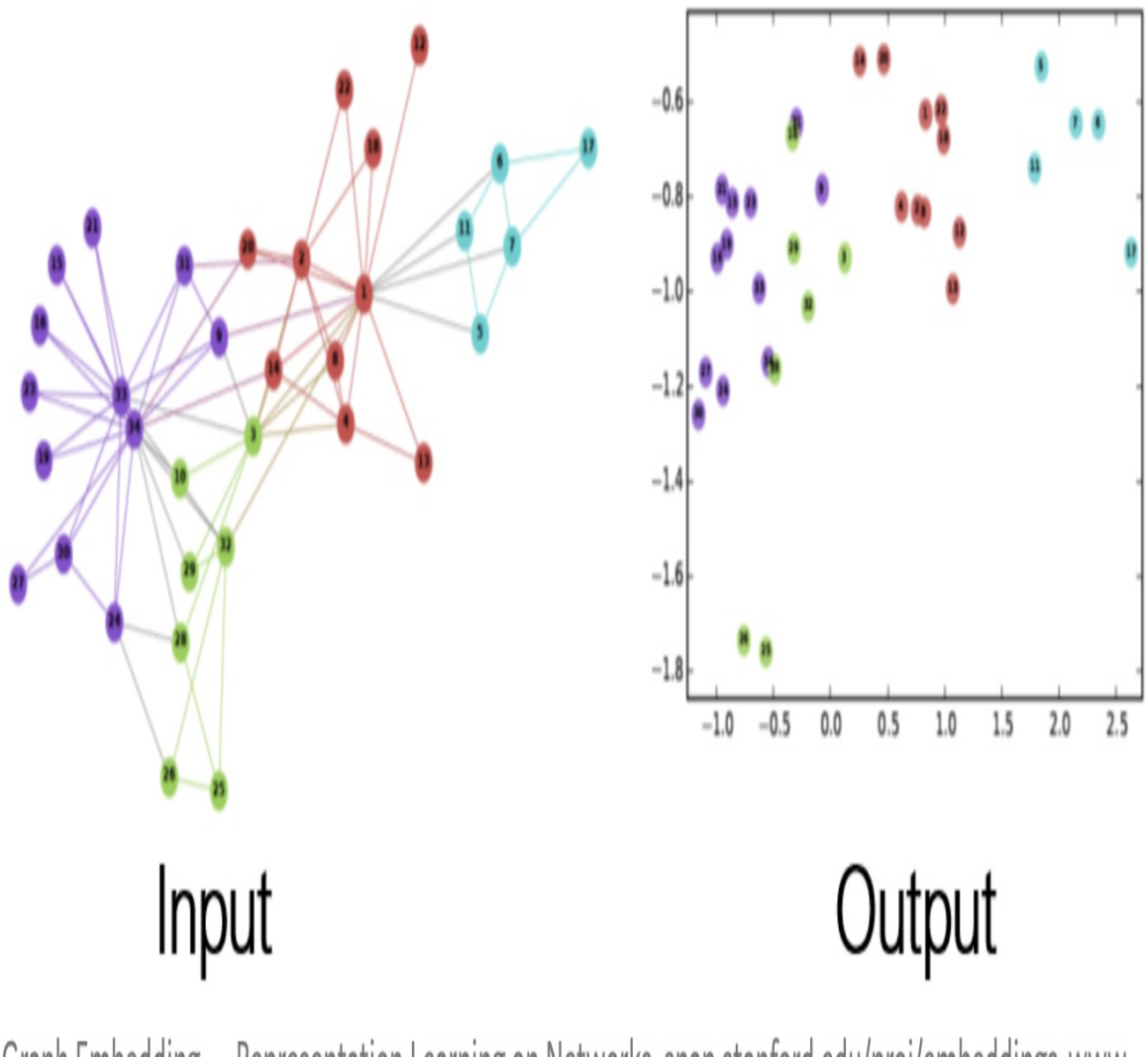
Graph algorithms and analytics have existed for many decades and been applied to many different use cases. Many of these methods are powerful, are used at scale, and have proven to be lucrative. PageRank, the first algorithm used by Google to order web search results, is probably the most famous example of a graph algorithm used to great success by an enterprise.

Applications where graph neural networks shine are problems that require predictive models. We either use the GNN to train such a model, or we use it to produce a representation of the graph data that can be used in a downstream model. We have already described many typical GNN predictive tasks including node-prediction,edge prediction, and graph prediction. We add one more task that GNNs excel at, graph representation. Let's get a clearer idea of these.

Graph Representation. The challenge of dealing with graphs, which are non-Euclidian in nature, is to represent them accurately in a way that can be input into a neural network environment. For GNNs, usually the first few layers are designed to do just that. These layers use embedding to transform the graph structure into a vector or matrix representation, or embedding. We'll discuss this in great detail in Chapter 3, with an example representation

in Figure 1.10.

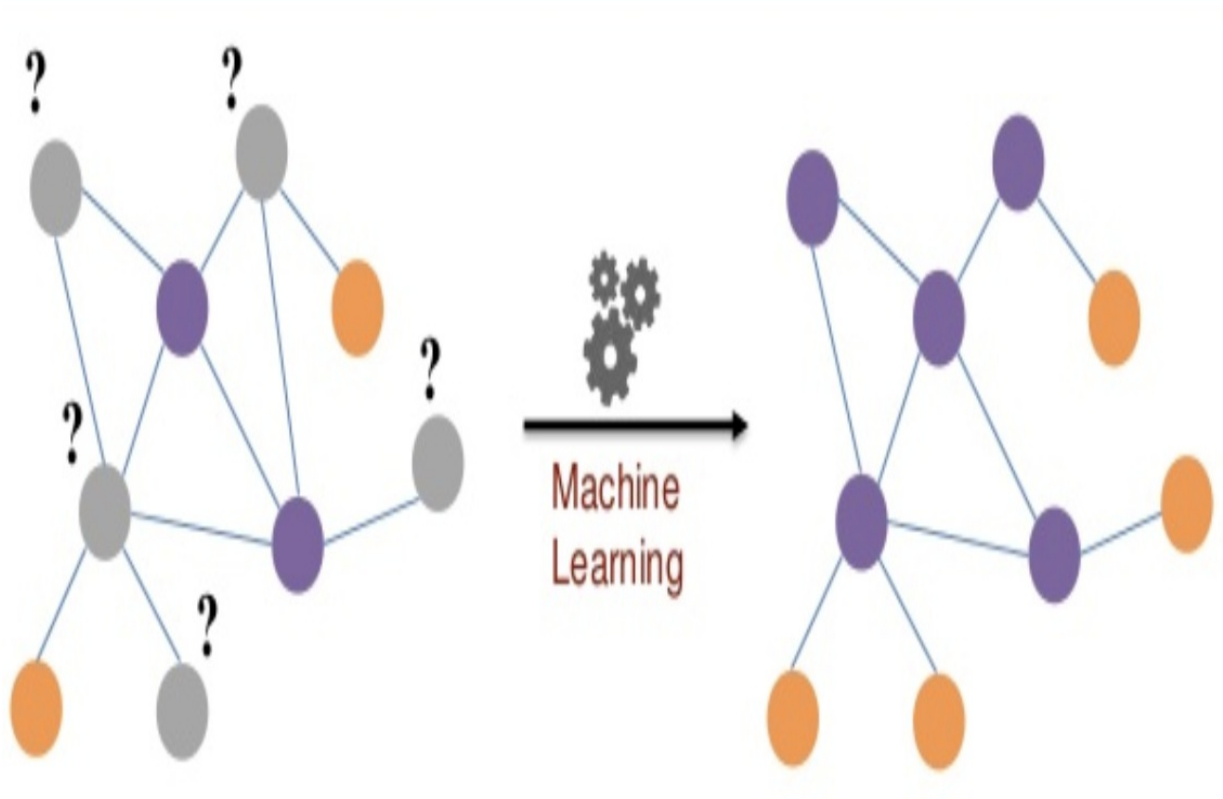
Figure 1.11 A visual example of a graph embedding. We start (left) with a non-Euclidean graph structure, and end up with an embedding (right) that can be projected in a 2D euclidean space (source: Chen et al.).



Node-Level Tasks. Nodes in graphs often have attributes and labels. Node classification is the node analog to traditional machine learning classification: given a graph with node classes, we aim to accurately predict the class of an unlabeled node. For example, we might use node classification if we want to

guess whether a passenger is family with another passenger or what board someone's Pinterest pin should belong to. We will learn to predict fraudulent users in financial networks in Chapter 5.

Figure 1.12 Node classification (source: snap-stanford).



Edge-Level Tasks. Edge-level tasks are very similar to those applied nodes. Link classification involves predicting a link's label in a supervised or semi-supervised way. However, we might also want to predict whether two nodes are correct when we don't have the full network. This is known as edge-prediction and describes when we want to infer whether there is an edge between nodes. We apply edge prediction to a consumer network in Chapter 6.

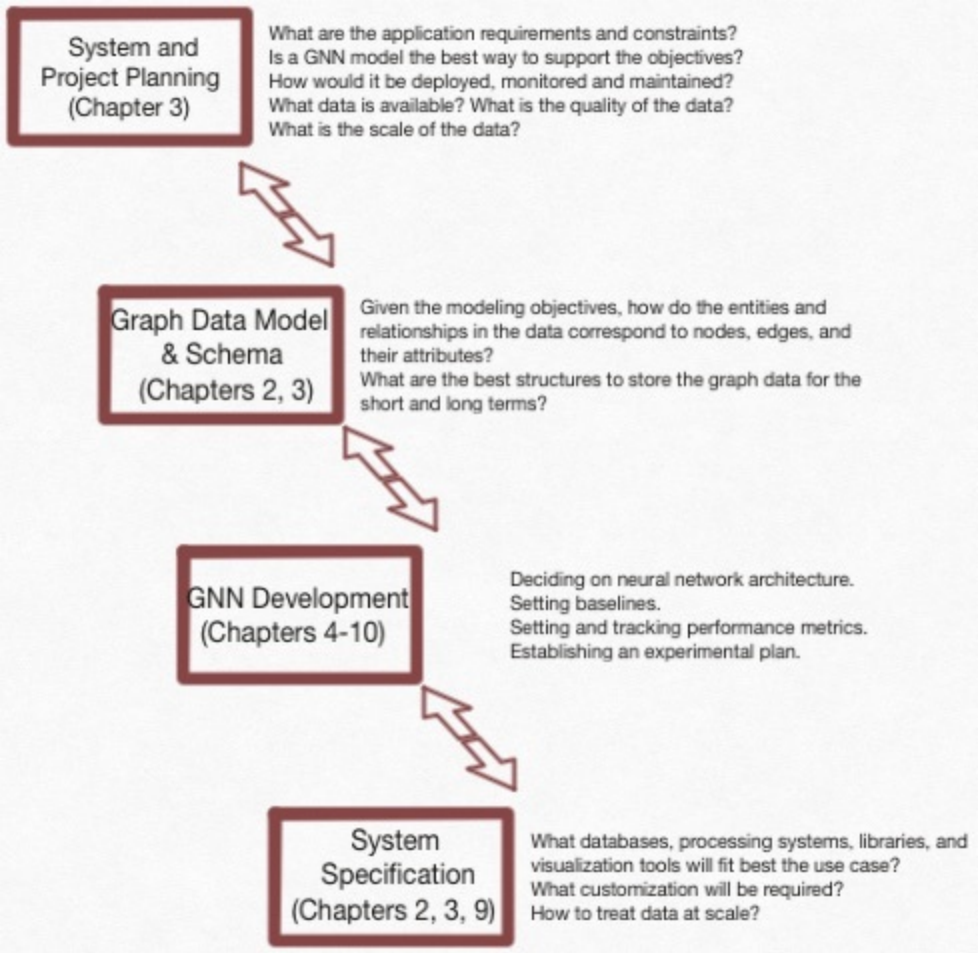
Graph-Level Tasks. In real scenarios, graphs may consist of a large set of connected nodes, many disconnected graphs, or a combination of a large connected graph with many smaller unconnected graphs. In these situations the learning task may be different. Graph classification and regression involve predicting the label or a quantity associated with the graph.

There are also unsupervised tasks that involve GNNs. These involve Graph Auto Encoders that embed graphs and do the opposite process of generating a graph from an embedding. We'll be describing generative models in great detail when we introduce the GAE architecture in Chapter 6.

1.6 The GNN workflow

So far we've covered graphs, GNNs, and some of the many tasks that GNNs can be used for. What's the big picture for the working practitioner? In this section, we will highlight two types of workflows that data scientists might use. One involves system planning, involving setting the project and infrastructure details required to create a successful GNN implementation for an application. This workflow, illustrated in Figure 1.13, is touched upon particularly in the first chapters.

Figure 1.13 The GNN Planning Workflow



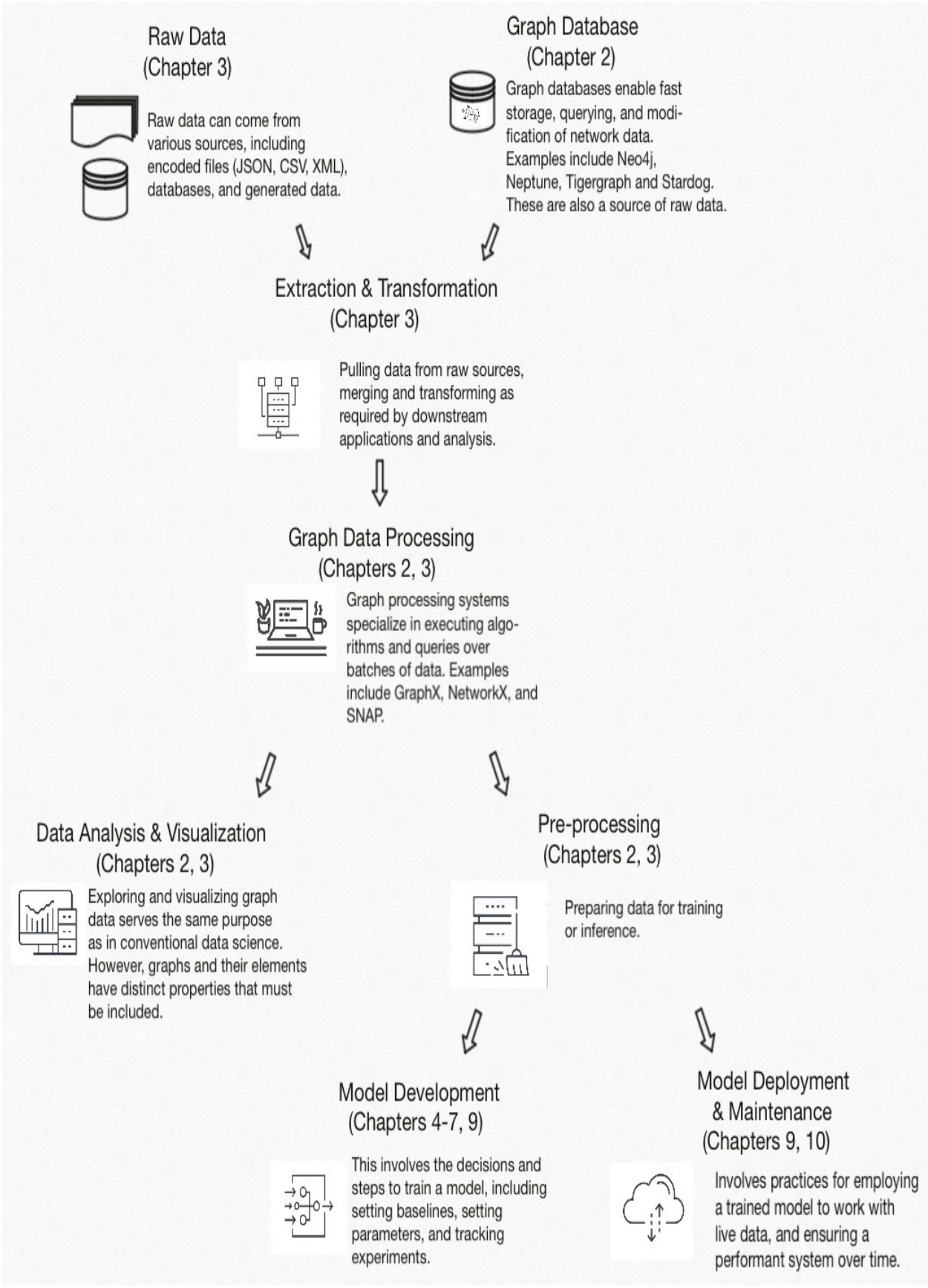
The second workflow involves the data and model development workflow, where we start with raw data and end up with a trained GNN model and its outputs. I've illustrated this below in stages related to the state and usage of the data. This is similar to, but not meant to be a data pipeline, which is an exact implementation of elements in this workflow.

Not all implemented workflows will include all of these steps, but many will draw from these items. At different stages of a development project, different parts of this process will typical be used. For example, when developing a model, data analysis and visualization may be prominent to make design decisions, but when deploying a model, it may only be necessary to stream raw data and quickly process it for ingestion into a model.

Though this book touches on the earlier stages in this workflow, the bulk of the book is on how to do model development and deployment of GNNs. When the other items are discussed, it is to clarify the role of graph data and

GNN models.

Figure 1.14 Graph Data Workflow. Starts as raw data, which is transformed into a graph data model that can be stored in a graph database, or used in a graph processing system. From the graph processing system (and some graph databases), exploratory data analysis and visualization can be done. Finally for graph machine learning, data is preprocessed into a form which can be submitted for training.


Raw Data
(Chapter 3)
Raw data can come from various sources, including encoded files (JSON, CSV, XML), databases, and generated data.
Graph Database
(Chapter 2)
Graph databases enable fast storage, querying, and modification of network data. Examples include Neo4j, Neptune, Tigergraph and Stardog. These are also a source of raw data.
Extraction & Transformation
(Chapter 3)
Pulling data from raw sources, merging and transforming as required by downstream applications and analysis.
Graph Data Processing
(Chapters 2, 3)
Graph processing systems specialize in executing algorithms and queries over batches of data. Examples include GraphX, NetworkX, and SNAP.
Data Analysis & Visualization
(Chapters 2, 3)
Exploring and visualizing graph data serves the same purpose as in conventional data science. However, graphs and their elements have distinct properties that must be included.
Pre-processing
(Chapters 2, 3)
Preparing data for training or inference.
Model Development
(Chapters 4-7, 9)
This involves the decisions and steps to train a model, including setting baselines, setting parameters, and tracking experiments.
Model Deployment
& Maintenance
(Chapters 9, 10)
Involves practices for employing a trained model to work with live data, and ensuring a performant system over time.

Let’s next examine this diagram from top to bottom.

First, we start with data in some initial state. Raw data refers to data which contains the information needed to drive an application or analysis, but has not been put into a form which can be ingested by an analytics tool. Such data can exist in databases, encoded files, or streamed data. For our purposes, graph databases are important data stores that we will emphasize.

In order to most effectively use this data and develop a data analytics model requires extraction, merging, and transformation of this raw data. This is shown in the next level of our diagram, where we explore the data, integrate key components of the data that may sit in different data stores, and prepare the data to be passed to the downstream model. One important tool at this stage is the graph processing system, which can provide a means to manipulate graph data as required.

At the third level, our workflow splits into two possible downstream applications: data exploration and visualization, and model training/inference. Data exploration and visualization tools allow engineers and analysis to draw useful insights and assessments of a dataset. For graph data, many unique properties and metrics exist that are key to such an assessment. Visualizing graph data and its properties also require special software and metrics.

For machine learning, we must preprocess our data for training and inference. This involves sampling, batching, and splitting (between train, validation, and test sets) the data. The GNN libraries we will study in this book, Pytorch Geometric and Deep Graph Library, have classes specially made for preprocessing. For graph data, we must take special care to format our data to preserve its structure when preprocessing it for loading into a model.

1.6.1 Applying the workflow: Health event prediction

In the following example, we’ll explore the frameworks and process we’ve just discussed. This example is meant to illustrate how GNN application development works at a high level and to point out what skills will be covered in this book.

Predicting harmful health events

Imagine you're the data scientist for a hospital or pensioners home. This state-of-the-art facility extensively monitors its patients with an array of sensors on the patients' bodies and in their rooms. Based on the data from these sensors, they have extensively cataloged a set of health status events that are common amongst the people in the hospital's charge.

Examples of such events are listed in table 1.2.

Table 1.2 Types of patient events that are logged.

Patient Physiological Events	Room Events	Social Events	Health Event
Increased blood pressure	Food provided	Social visit	Heart Attack
Neurological condition	Lights turned off	Difficulty speaking	Stroke
Impaired motility	TV turned on	Patient undertakes daily exercise with physical therapist	Falling

The chief doctor knows that certain events can indicate potentially serious health events (e.g. a bad fall or even strokes). Thus, the doctor wants you to train a model that can use the log data to identify such events. Not the serious negative events themselves, but 'prelude' events that indicate a more serious occurrence might be happening in the near future. This model will be used to power an alert application in the hospital's existing management system.

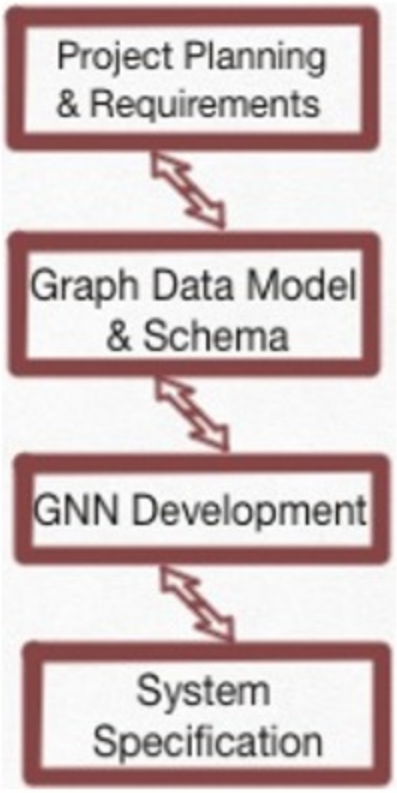
Given the importance of relationships in this problem, you think a GNN model may provide some advantages.

With the problem defined, let's focus the example on the development issues most relevant to graphs and GNNs. Here we will cover the four major steps in the GNN planning workflow described earlier :

- Project and System Planning
- Specifying a Graph Data Model from Raw Data
- Defining the GNN problem and approach
- Data Pipelines for Training and Inference

For a visual guide to this process, we can borrow from Figure 1.12, reproduced here as Figure 1.14:

Figure 1.15 The four phases we'll go through to build the GNN for predicting preclude health events



Project planning

Scale of Facility. The medical facility has an average of 500 patients and an average of 5 sensors per patient. The system can collect 5000 data points per second with low latency.

Data Systems. The hospital employs an IoT system to ingest and parse data produced by the sensor devices. This data is streamed to a relational database. Every 24 hours, the data in this first database is transferred to longer term storage built for analytics.

For training, data will be pulled from the analytics system. Upon deployment, for real-time alerts, your model will have to make predictions using the streamed data. The time to make a prediction should be fast, so that the alerts can be effectively used by the medical staff. Predictions will have to be made at scale simultaneously.

Representing the data as a graph

Given the doctor's hypothesis, you examine the data and propose a graph representation that can be used by a model.

You consider each log and investigate translating the data into a relational format. Logs are kept by the minute for each patient. A query of log information for 3 patients over a few minutes looks like this:

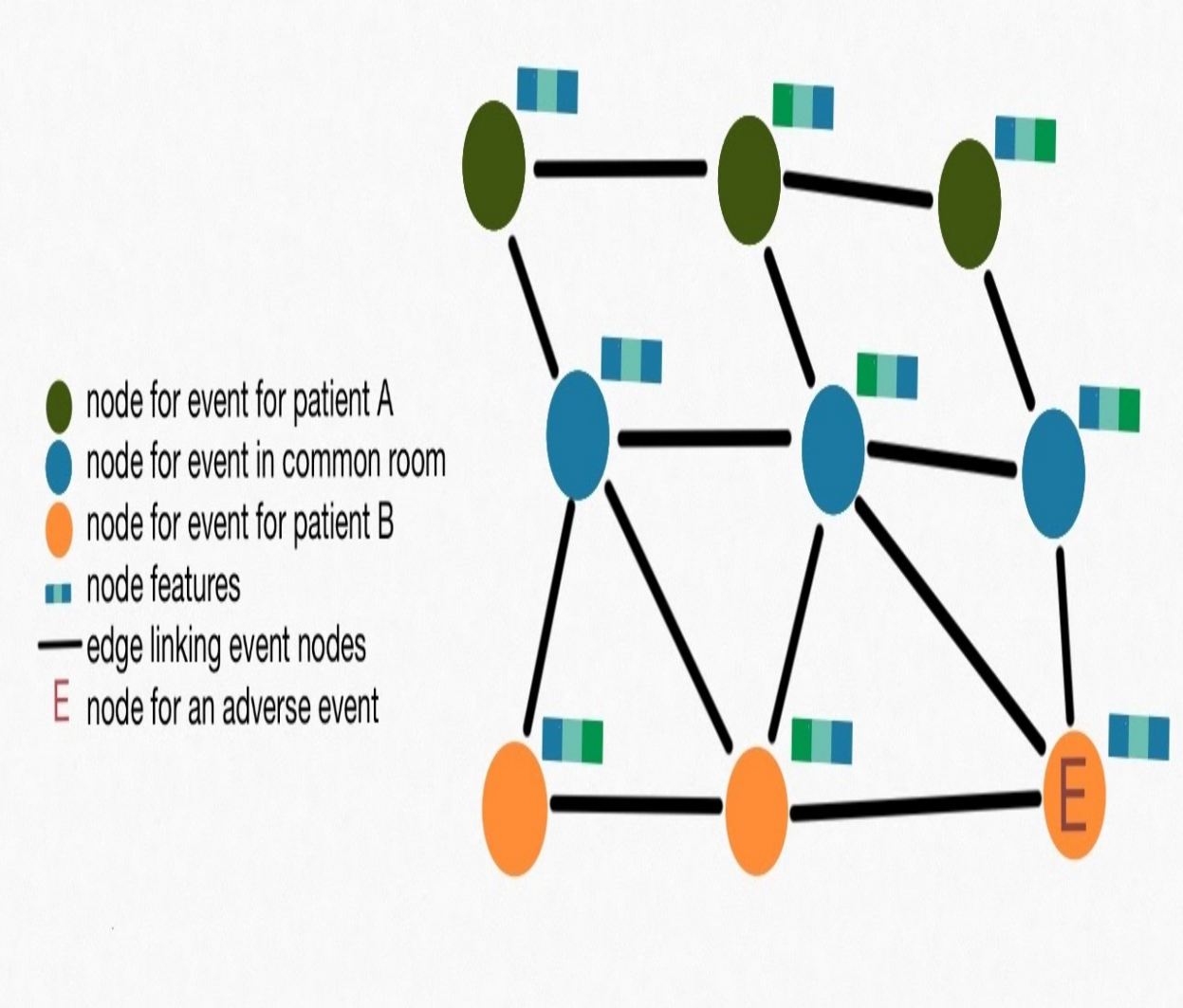
Table 1.3 An excerpt from a daily log.

TimeStamp	Patient ID	Event	Location ID
9:00:00 Dec 3rd	Null	Door Opens	Room A
9:02:00 Dec 3rd	1	Breakfast is served	Room A
9:04:00 Dec 3rd	Null	Door Opens	Room A

9:19:00 Dec 3rd	1	Begin Sleep	Room A
9:30:00 Dec 3rd	2	Room TV Turned On	Room C
9:36:00 Dec 3rd	3	Patient begins exercise	Room A
9:40:00 Dec 3rd	2	Patient Sneezes	Room C

As you study the data, you notice a few things. First, you realise that each event/observation in the query results can be interpreted as a node in a graph. Then you can consider the columns in each row observation can be used as node features. Finally, you see that certain sets of events happen in time and space proximity to one another. This proximity can be interpreted as edges between the event nodes. An example would be when two patients share a room. Events of the room, and for each patient that happen near each other in time would have edges between them. A third patient in a room on another floor would not have linked events. Also, events that happen more than several minutes apart would not be directly linked. Hence you represent the data as a graph that describes all these different components, which we show in Figure 1.16.

Figure 1.16 A graph representation of our medical data.



Defining the machine learning task

From your initial data inspection and problem formulation, you have decided to treat this as a supervised classification problem, where events are classified as leading to negative health outcomes or not. Though you think a GNN would be a great potential inference engine for this problem, you establish baselines with simpler models. These might include

- Simple logistic regression
- Random forest where individual events are classified
- Time-series classification, where selected sequences of events are classified

Then, for the GNN model, we have the following characteristics:

- Supervised Node prediction task using the data modelled as a graph
- The GraphSage architecture for node classification, which we shall learn about in Chapter 4.

System Specification

Finally, we can make explicit some of the decisions that you have made for the system specification.

- Data Extraction and Transformation. Important here are the processes and systems used to transform the IoT data to graph data, following the graph data model set above. Since the training data is drawn from a longer term data store, and the live data will be streamed, care must be taken to make the transformed data consistent. Furthermore, the problem uses health data so you make sure that patient privacy is respected at all steps.
- GNN Library and Ecosystem. At present, there are few choices for the GNN library, the most prevalent being Deep Graph Library and Pytorch Geometric. Factors influencing this choice would depend upon the ease of use and learning curve; need for support of training and inference on large scale data; and training time and cost.
- Experimental Design. In developing the GNN, factors in the experimental design would include the graph data model, as well as input features, parameters and hyperparameters. For example, the size limits of the graphs submitted for inference (e.g., number of nodes, or number of edges) may have an impact on the GNN performance.

The above case study gives a flavor for the types of things to consider when undertaking any machine learning project, but especially one that might require graph-based learning. We have only given an overview for some of the choices you might face. Furthermore, we note that many choices might change over time as more data becomes available or certain methods underperform. However, having these questions and answers made explicit before undertaking the project is important and we strongly recommend you try it with your next, or current, machine learning project.

1.7 Summary

- Graph Neural Networks (GNNs) apply deep learning methodologies to network (also called graph) data structures.
- GNNs came about due to modern techniques that allowed neural networks to learn from non-Euclidean data structures.
- Graph neural networks have a wide potential use space, since most systems in the real world have non-Euclidean geometries.
- GNNs have made an impact in the physical sciences, knowledge and semantics, and have been applied to
- Much of the academic work and developed frameworks have been based upon relatively small systems. Only recently have some techniques been introduced that can be applied to large scale production systems.

1.8 References and further reading

Academic Overviews of GNNs

Hamilton, William, *Graph Representational Learning*, Morgan & Claypool, 2020.

Bronstein, Michael M. “Bronstein, Michael M., et al. "Geometric deep learning: going beyond euclidean data." *IEEE Signal Processing Magazine* 34.4 (2017): 18-42.”

Zhou, Jie, *Introduction to Graph Neural Networks, Morgan & Claypool, 2020.*

Wu, Zonghan, et al. "A comprehensive survey on graph neural networks." *IEEE transactions on neural networks and learning systems* (2020).

Chen, Fenxiao, Yun-Cheng Wang, Bin Wang, and C-C. Jay Kuo. "Graph representation learning: a survey." *APSIPA Transactions on Signal and Information Processing* 9 (2020).

References to Section 1.2 Examples

Ying, Rex, et al. "Graph convolutional neural networks for web-scale recommender systems." *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2018

Sanchez-Gonzalez, Alvaro, et al. "Graph networks as learnable physics engines for inference and control." International Conference on Machine Learning. PMLR, 2018.

Sanchez-Lengeling, Benjamin, et al. "Machine learning for scent: Learning generalizable perceptual representations of small molecules." arXiv preprint arXiv:1910.10685 (2019).

Academic Overviews of Graphs

Deo, Narsingh, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall of India, 1974.

2 System Design and Data Pipelining

This chapter covers

- Architecting a graph schema and data pipeline to requirements
- Working with various raw data sources and transforming them for training
- Taking an example dataset from raw data through preprocessing
- Creating datasets and data loaders with Pytorch Geometric

In the last chapter, we gave an overview for graphs and graph-based learning. However, we want this book to be helpful for the data scientist who is looking to practically implement machine learning tasks as well as those new to the field of graph-based ML. We covered a few of the main principles behind designing machine learning systems and building data pipelines in the last chapter, such as system specification, project planning and model development. However, developing ML systems designed for graph data requires additional considerations. This chapter will explain some of these considerations, including turning tabular data into graphs and preprocessing with a suitable graph-based ML package. In this chapter, along with the rest of the book, we'll be using PyTorch Geometric as our package of choice.

We'll proceed as follows. First, we'll discuss some of the main choices for graph-based ML system planning, such as working out project objectives and scope and what types of questions to keep in mind. Once we have a clear outline of the project, we'll discuss how to get the data ready for use in the project by walking through an example data pipeline from raw data to preprocessing. Once both of these steps are completed, we can progress to the development of our ML models. This is covered in a problem-specific way in Chapters 3 to 7. Finally, in Chapter 8, we'll discuss how to scale these models for larger projects.

In order to better understand how to implement a GNN-based analytics

pipeline, we'll be using social data. Social networks are great examples of relational data and are a common application domain for GNNs. For example, they can be used to predict connections, such as LinkedIn's recommendation functionality, or to infer missing details about a user to improve their experience of a service.

We'll be considering the toy problem of a recruitment firm who are looking to help both their recruiters and employees. We'll set out how this can be recast as graph-based problem and how to prepare for an ML project on this data. Our objective in this chapter is to create a simple data workflow that will take our data from its raw format, perform exploratory analysis, and preprocess it ready for GNN training in the next chapter.

Where is the code?

Code from this chapter can be found in notebook form at the github repository and in Colab. Data from this chapter can be accessed in the same locations.

2.1 Graph Example: A Social Network

We're going to begin by considering the case of a hypothetical recruiting firm,called Whole Staffing. Whole Staffing hires employees for a number of different industries and maintains a database of all their candidate profiles, including their history of engagement with past firms and whether candidates have given or made referrals to other current or prospective candidates.

Whenever an existing job candidate refers someone to the firm, this referral is logged, noting both the person who has been referred and the person who is the referee. Sometimes multiple people will refer the same person, especially if the candidate is very suitable for the role or if they excel. On the other hand, referrals might be already in the database and, in this case, referrals are ignored. Hence, at any point in time, all the referrals create a single large social network of referees and referrals. We're going to ignore who refers who so that the graph is undirected. We'll also be ignoring whether there are repeat referrals, so that relationships between candidates remain unweighted.

Whole Staffing wants to get the most value from their database and have brought you on as a data scientist to help them do this. They have a few initial questions about their collection of job candidates:

1. Some profiles have missing data. *Is it possible to fill in missing data without bothering the candidate?*
2. History has shown that candidates that worked well in the past, do well on future teams. Thus, when companies are hiring teams of workers, Whole Staffing would like to refer groups of candidates who know each other. *Is it possible to figure out which candidates have worked together in the past, even if they have not disclosed any relationships with past colleagues?*

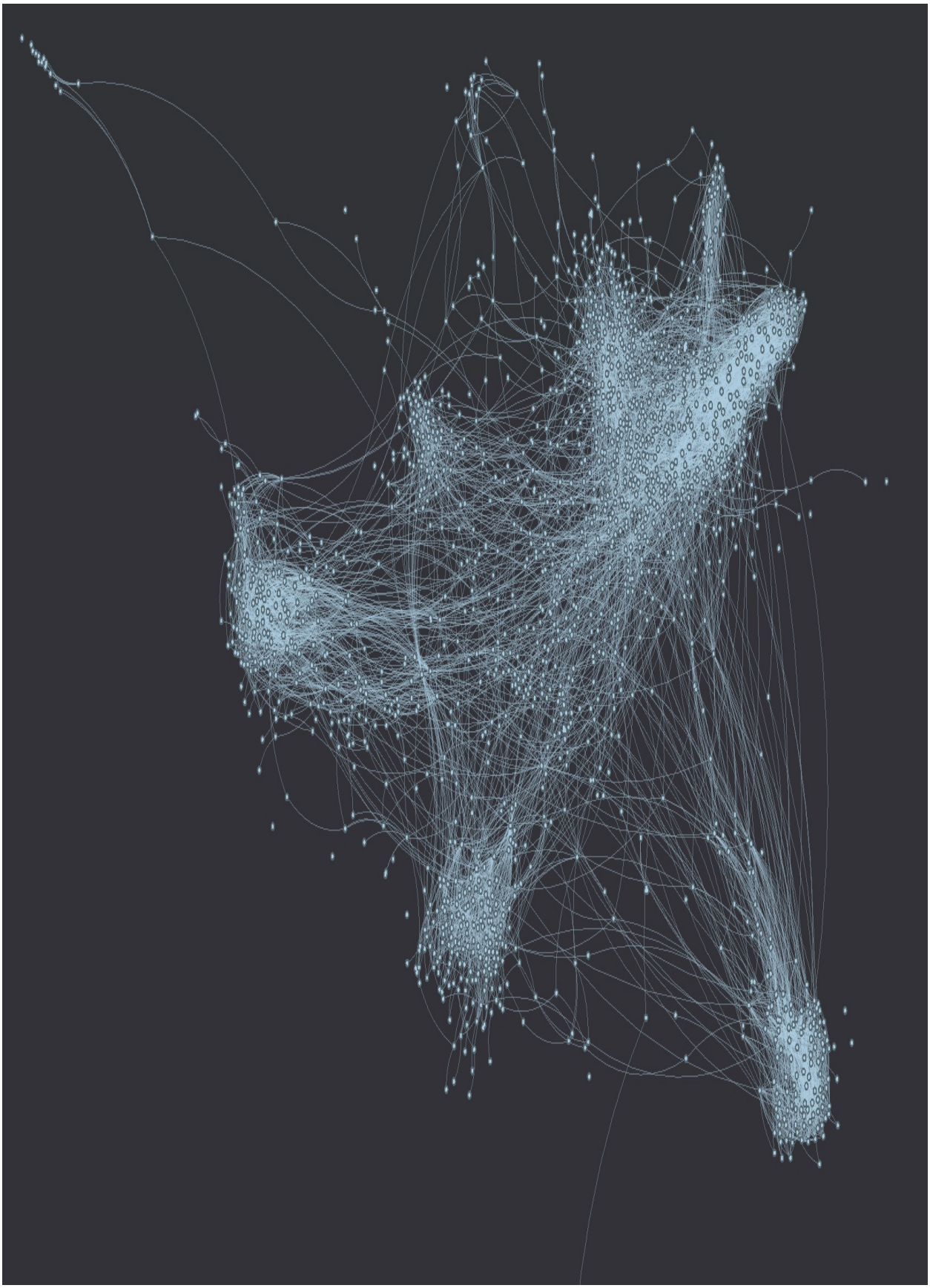
Whole Staffing have tasked you with mining the candidate data to answer and report on these and other questions. Amongst other analytical and machine learning methods, you think there may be an opportunity to represent the data as a network and use a GNN to answer these questions. To get started, you need to come up with a workflow that starts with the raw candidate data, and preprocesses it so it can be fed into a GNN model.

We're going to walk through the steps in this process. First, we have to make some early decisions about how we'll handle our data, what graph data model we might use, and what pipeline systems best fit our problem.

The way to make these decisions is to first examine the raw data. In Appendix A, we've provided some initial information about this professional social network data along with information on how you can load and play around with it yourself. While we'll be repeating this information through this chapter, as it becomes helpful, we strongly recommend that you download and explore the data yourself too.

In Figure 2.1, we can see an example graph that is formed from the tabular data. We can see that it is rich in information, with some central clusters of closely connected individuals as well as many individuals with fewer connections. We're going to assume that the edges are undirected, so we don't distinguish between whether someone is referred or is a referee and unweighted, so that all edges are given the same importance.

Figure 2.1 Visualization of Whole Staffing's employee database as a graph.



After performing our initial data exploration, we have found out that the node with the highest number of referrals, measured as the diameter of the largest component, is 10, that there are 1933 nodes and more than twelve thousand edges. We've also set this information out in Table 2.1, using terms explained in Appendix A. This type of exploratory data analysis is always a good idea when first preparing for a new project. Understanding how many nodes and edges are in the graph help us to work out the scale that we need to work with. Similarly, identifying whether the graph is directed/undirected and/or weighted/unweighted has downstream impacts on what GNN models we might want to use. Finally, it's always a good idea to become familiar with your data to develop some personal intuition of what kind of results you might expect. For example, if we later were to see that our model predicts many employees end up with more than ten referrals, then we might want to go back and check that our model is performing accurately, given the largest component was ten in the data.

Table 2.1 Graph features of the dataset.

Number of Nodes	1933
Number of Edges	12239
Type of Graph	Undirected, Unweighted
Diameter of Largest Component	10

2.2 GNN system planning

Now that we have some initial impressions of the data and an idea of that type of problems that we want to address, we can proceed to planning our ML system. As a reminder, we want to know if we can train a GNN model on

recruitment data (represented in graph format) to answer some queries about job candidates in the professional social network. The first two questions are "Can we train our model to fill in missing information about individual candidates?" and "Can we get the model to find missing relationships between candidates?" If you were paying attention in Chapter 1, you might already have an idea of the type of graph-based tasks these relate to.

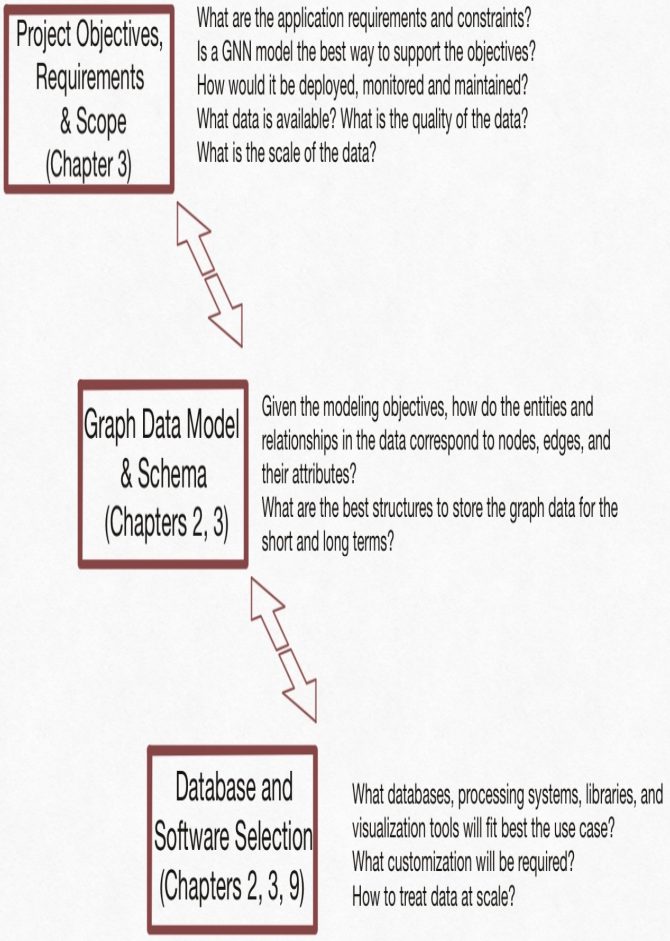
First, for missing information about candidates, we'll be trying to work out information about individual nodes. For example, suppose a candidate forgot to update their profile when they changed jobs. However, all their nearest neighbors in the social network work in a specific industry such as manufacturing. We can then predict that this candidate has moved into manufacturing and suggest new roles to them accordingly. Hence, this is a *node-prediction task.*

The second question uses information about the nodes as well as information about the graph topology, contained in the edges, to predict whether we should add new edges. For example, suppose three candidates all give referrals to each other for many different roles. However, none of these candidates are known to have worked together despite being in the same industry and city. This might imply that they have worked together in the past and our employment graph should be updated accordingly. Hence, this is an *edge-prediction* task.

Whether a task is node-prediction or edge-prediction or something else, such as graph-prediction, will impact choices on our model selection later on. However, more broadly, understanding exactly what kind of prediction task we need to do can be helpful to clearly frame the problem, both to ourselves and employers. While graphs are natural ways to understand data, tabular formats have been dominant for some time. Understanding what graph-based tasks we'll be undertaking helps to motivate why recasting our data as a graph is useful and not unnecessary time spent.

For the rest of this chapter, we'll be following the process in Figure 2.2, which we first introduced in Chapter 1. We'll be talking about high-level project planning, schema creation, and tool selection. Then we'll give information about the practicalities of loading and dealing with graph data.

Figure 2.2 Overview of Graph System Planning. Each of the elements are interdependent and will be discussed in the following sections.


Project Objectives, Requirements & Scope (Chapter 3)
What are the application requirements and constraints?
Is a GNN model the best way to support the objectives?
How would it be deployed, monitored and maintained?
What data is available? What is the quality of the data?
What is the scale of the data?
Graph Data Model & Schema (Chapters 2, 3)
Given the modeling objectives, how do the entities and relationships in the data correspond to nodes, edges, and their attributes?
What are the best structures to store the graph data for the short and long terms?
Database and Software Selection (Chapters 2, 3, 9)
What databases, processing systems, libraries, and visualization tools will fit best the use case?
What customization will be required?
How to treat data at scale?

2.2.1 Project objectives and scope

Project objectives and scope relates to the high-level goals and objectives of a project, including its requirements and scope. Let's discuss each of these for our toy example of helping a recruitment firm.

Project Objectives. The directions and decisions taken in a project should be most influenced by the core objectives of the project or application. Having a well-defined objective up front will save a lot of headaches down the line.

Project Requirements and Scope. There can be many pertinent requirements that have a direct impact on your project. For example, we might consider

- **Data Size and velocity**: what is the size of the data, in terms of item counts, size in bytes, or number of nodes? How fast is new information added to the data, if at all? Is data expected to be uploaded from a real-time stream, or from a data lake that is updated daily?
- **Inference Speed**: How fast are the application and the underlying machine learning models required to be? Some applications may require sub-second responses, while for others, there is no constraint on time.
- **Data Privacy.** What are the policies and regulations regarding personally identifiable information (PII), and how would this involve data transformation and pre-processing?

In our social graph example, we have set the objective of filling in missing candidate information. For Whole Staffing, the deliverable for this objective will be an application that can periodically (say, bi-weekly) scan the candidate data for missing items. Missing items will be inferred and filled in.

Some of the requirements for this could be:

- Data Size: Greater than 1933 candidate profiles, which we learnt from our data exploration step. Less than 1MB
- Inference Speed: Application will run bi-weekly, and can be completed overnight so we don't have a considerable speed constraint.
- Data Privacy: No personal data that directly identifies a candidate can be

used. However, data known to the recruitment company, such as whether employees have been successfully placed at the same employer or have referred other works can be used.

2.2.2 Designing graph data models and schema

In machine learning problems involving tabular data, images or text, our data is organized in an expected way, with implicit and explicit rules. For example, when dealing with tabular data, rows are treated as observations, and columns as features. We can join tables of such data by using indexes and keys. This framework is flexible and relatively unambiguous. We may quibble about which observations and features to include, but we know where to place them.

When we want to express our data with graphs, in all but the simplest scenarios, we have several options for what framework to use. With graphs, it's not always intuitive where we place the entities of interest. It's the non-intuitiveness and ambiguity that drives the need for defining explicit data models for our graph data.

By actively choosing and documenting our graph data model up front, we can avoid technical debt and more easily test the integrity of our data. We can also experiment more systematically with different data structures.

Technical debt can occur when we have to change and evolve our data, but haven't planned for backward- or forward-compatibility in our data models. It can also happen when our data modeling choices aren't a good fit for our database and software choices, which may call for expensive (in time or money) workarounds or replacements.

Having well defined rules and constraints on our data models give us explicit ways to test the quality of our data. For example, if we know that our nodes can at most have two degrees, we can design simple functions to process to test every node against this criteria.

Also, when the structure and rules of our graphs are designed explicitly, it increases the ease with which we can parameterize these rules and experiment with them in our GNN pipeline.

In this section, we'll talk about the landscape of graph data model design, and give some guidelines on how to design your own data models. Then we'll use these guidelines in thinking about our social graph example. This section is drawn from several references, listed at the chapter's end.

Data Models and Schemas

In Chapter 1, we were introduced to graph data models, which are ways to represent real world data as a graph. Such models can be simple, consisting of one type of node and one type of edge. Or they can be complex, involving many types of nodes and edges, metadata, and, in the case of knowledge graphs, ontologies.

Data models are good at providing conceptual descriptions of graphs that are quick and easy to grasp by others. For people who understand what a property graph or an RDF graph is, telling them that a graph is a bi-graph implemented on a property graph can reveal much about the design of your data (property graphs and RDF graphs are explained in Appendix A).

In general, a **schema** is a blueprint that defines how data is organized in a data storage system. In a sense, a graph schema is a concrete implementation of a graph data model, explaining in detail how the data in a specific use case is to be represented in a real system. Schemas can consist of diagrams and written documentation. Schemas can be implemented in a graph database using a query language, or in a processing system using a programming language.

A schema should answer the following questions:

- What are the elements (nodes, edges, properties), and what real world entities and relationships do they represent?
- Does the graph include multiple types of nodes and edges?
- What are the constraints around what can be represented as a node?
- What are the constraints around relationships? Do certain nodes have restrictions around adjacency and incidence? Are there count restrictions for certain relationships?
- How are descriptors and metadata handled? What are the constraints on

this data?

Depending on the complexity of your data and the systems in use, one may use multiple, but consistent schemas. A **conceptual schema** lays out the elements, rules and constraints of the graph, but is not tied to any system. A **system schema** would reflect the conceptual schema's rules, but for a specific system, such as a database. A system schema could also omit unneeded elements from the conceptual schema

Depending on the complexity of the graph model and the use cases, one or several schemas could be called for. In the case of more than one schema, compatibility between the schemas via a mapping must also be included.

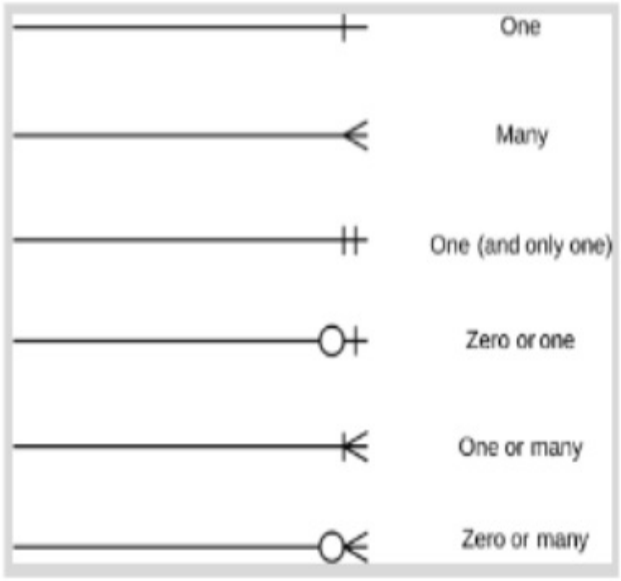
For graph models with few elements, rules and constraints, a simple diagram with notes in prose is probably sufficient to convey enough information to fellow developers and to be able to implement this in query language or code.

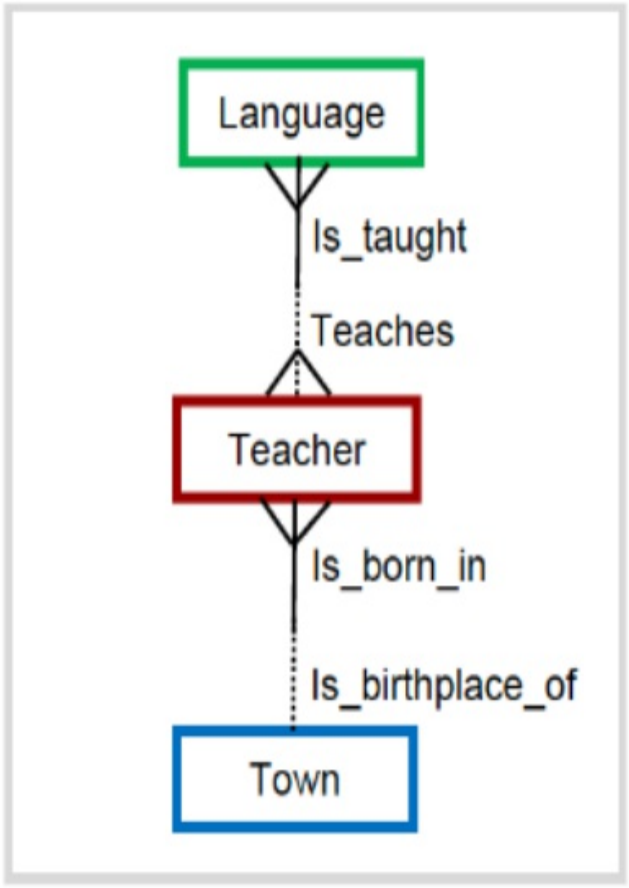
For more complex network designs, entity-relationship diagrams (ER diagrams, or ERDs) and associated grammar are useful in illustrating network schemas in a visual and human readable way. In the next few sections, we'll go through ERDs in more detail.

Entity-Relation (E-R) Diagrams

E-R diagrams have the elements to illustrate a graph's elements (nodes, edges, and attributes) and the rules and constraints governing a graph. Figure 2.3 (top) shows some connectors notation that can be used to illustrate edges and relationship constraints. Figure 2.3 (bottom) shows an example of a schema diagram that conveys three node types that might be represented in our recruitment example (employee, referee, and business), and two edge types ('to refer', and 'to be employed by). The diagram conveys implicit and explicit constraints.

Figure 2.3 At top is the relationship nomenclature for E-R diagrams. The bottom diagram is an example of a conceptual schema using an E-R diagram.





Some explicit constraints are that one employee can refer many other

employees and that one referee can be referred by many employees. Another explicit constraint is that a person can only be employed full-time by one business, but one business might have many employees. An implicit constraint is that, for this graph model, there can be no relationship between a business and a referral.

To not start from scratch in developing a graph data model and schema for your problem, there are several sources of published models, and schemas. They include industry standard data models, published datasets, published semantic models (including knowledge graphs), and academic papers. A set of example sources is provided in Table 2.2.

These examples are great ways to benchmark and learn how to implement many of the different models we'll be describing in Chapters 3 to 7. They cover a huge range of different graph topics, including molecules, social networks, knowledge graphs, and many others. We've also given some example problem domains in Table 2.2 to help you explore these data.

Public graph datasets also exist in several places. Published datasets have accessible data, with summary statistics. Often however, they lack explicit schemas, conceptual or otherwise. To derive the dataset's entities, relations, rules and constraints, querying the data becomes necessary.

For semantic models based on property, RDF and other data models, there are some general ones, and others that are targeted to particular industries and verticals. Such references seldom use graph-centric terms (like *node*, *vertex*, and *edge*), but will use terms related to semantics and ontologies (e.g., *entity*, *relationship*, *links*). Unlike the graph datasets, the semantic models offer data frameworks, not the data itself.

Reference papers and published schemas can provide ideas and templates that can help in developing your schema. There are a few use cases targeted toward industry verticals that both represent a situation using graphs, and use graph algorithms, including GNNs, to solve a relevant problem. Transaction fraud in financial institutions, molecular fingerprinting in chemical engineering, page rank in social networks are a few examples. Perusing such existing work can provide a boost to development efforts. On the other hand, often such published work is done for academic, not industry goals. A

network that is developed to prove an academic point or make empirical observations, may not have qualities amenable to an enterprise system that must be maintained and be used on dirty and dynamic data.

Where to Find Graph Data

I want to briefly outline where to get graph data.

From Non-Graph Data. In the above sections, I have assumed that the data lies in non-graph sources, and must be transformed into a graph format using ETL and preprocessing. Having a schema can help guide such a transformation and keep it

Existing Graph Data Sets. The number of freely available graph datasets is growing. Two GNN libraries we use in this book, DGL and Pytorch Geometric, come with a number of benchmark datasets installed. Many such datasets are from the influential academic papers. However, such datasets are small scale, which limits reproducibility of results, and whose performance don't necessarily scale for large datasets.

A source of data which seeks to mitigate the issues of earlier benchmark datasets in this space is Stanford's Open Graph Benchmark (OGB). This initiative provides access to a variety of real world datasets, of varying scales. OGB also publishes performance benchmarks by learning task.

Table 2.2 in section 2.2.2 lists a few repositories of graph datasets.

From Generation. Many graph processing frameworks and graph databases allow the generation of random graphs using a number of algorithms. Though random, depending on the generating algorithm, the resulting graph will have characteristics that are predictable.

Table 2.2 Graph datasets and semantic models.

Source	Type	Problem domains	URL

Open Graph Benchmark	Graph Datasets and Benchmarks	Social networks, drug discovery	https://ogb.stanford.edu/
GraphChallenge Datasets	Graph Datasets	..	https://graphchallenge.mit.edu/data-
Network Repository	Graph Datasets	..	http://networkrepository.com/
SNAP Datasets	Graph Datasets	..	http://snap.stanford.edu/data/
Schema.org	Semantic Data Model	..	https://schema.org/
Wikidata	Semantic Data Model	..	https://www.wikidata.org/
Financial Industry Business Ontology	Semantic Data Model	Finance	https://github.com/edmcouncil/fibo
Bioportal	List of Medical Semantic Models	Medical	https://bioportal.bioontology.org/ont

2.2.3 Social Graph Example

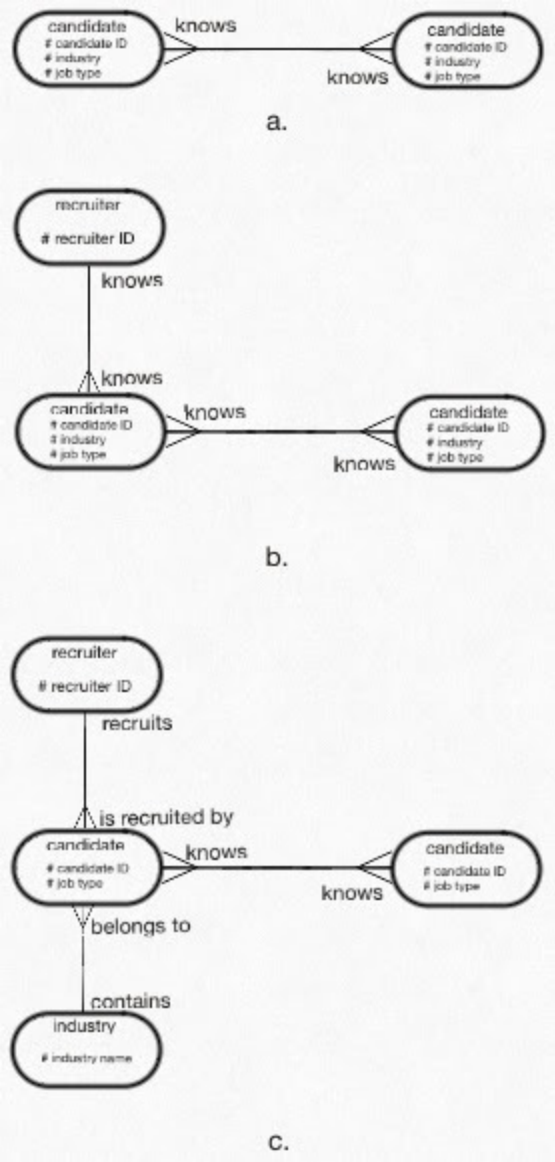
To design a conceptual and system schemas for our example dataset, we should think about:

- The entities and relationships in our data
- Possible rules and constraints
- Operational constraints, such as the databases and libraries at our disposal
- The output we want from our application

Our databases consist of candidates and their profile data (e.g., industry, job type, company, etc), recruiters. Properties can also be treated as entities; for instance, ‘medical industry’ could be treated as a node. Relations could be: ‘candidate knows candidate’, ‘candidate recommended candidate’, or ‘recruiter recruited candidate’.

Given these choices, we have a few options for the conceptual schema Option A is shown in Figure 2.4.

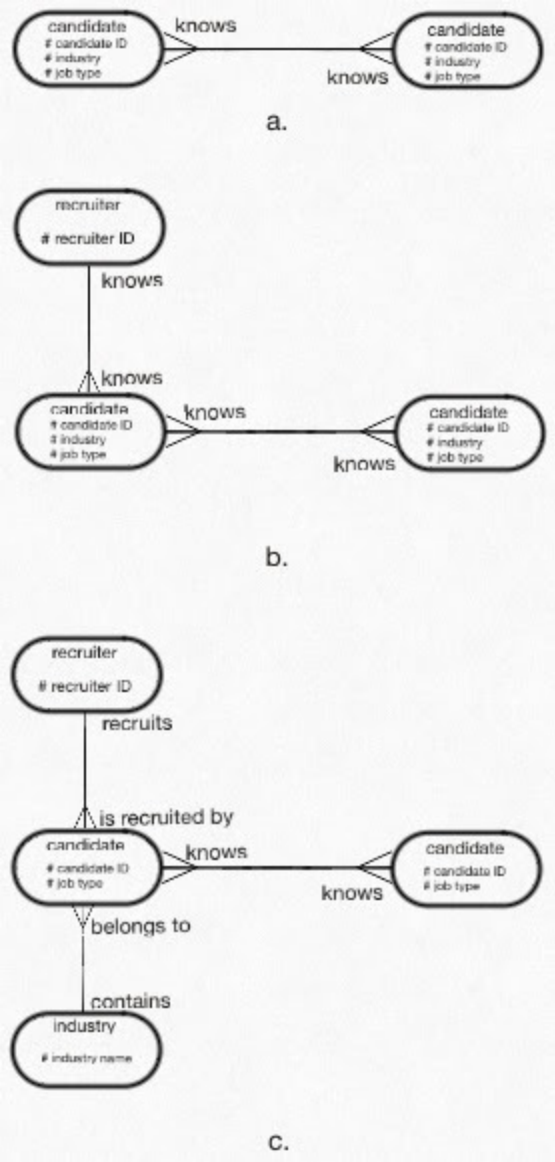
Figure 2.4 Schema with one node type and one edge type



As you can see, example A consists of one node type (*candidate*) connected by one undirected edge type (*knows*). Node attributes are the candidate's *industry* and their *job type*. There are no restrictions on the relationships, as any candidate can know 0 to *n*-1 other candidates, where *n* is the number of candidates.

The second conceptual schema is shown in Figure 2.5.

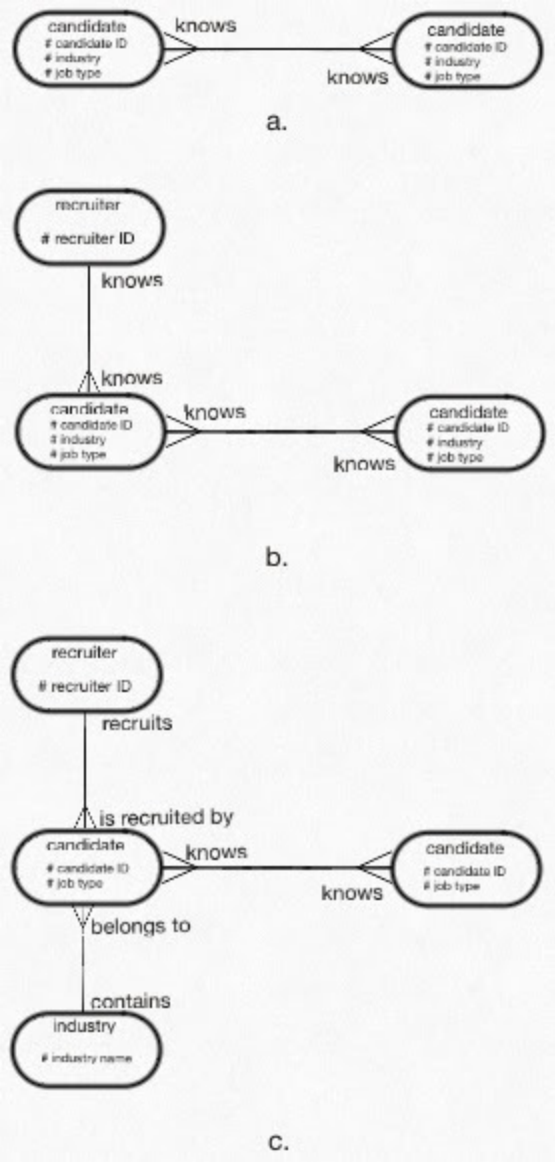
Figure 2.5 Schema with two node types and one edge type



Example B consists of two node types (*candidate* and *recruiter*), linked by one undirected edge type (*knows*). Edges between candidates have no restrictions. Edges between candidates and recruiters have a constraint: a candidate can only link to one recruiter, while a recruiter can link to many candidates.

The third schema, shown in Figure 2.6, has multiple node and relationship types.

Figure 2.6 Schema with three node types and three edge types



In Example C, the types are *candidate*, *recruiter*, and *industry*. Relation types include *candidate knows candidate*, *recruiter recruits candidate*, *candidate is a member of industry*. Note, we have made *industry* a separate entity, rather than an attribute of a candidate. These types of graphs are known as *heterogeneous,* as they contain many different types of nodes and edges. In a way, we can imagine these as multiple graphs that are layered on top of each other. When we have only one type of nodes and edges, then graphs are known as *homogenous.* Some of the constraints for Example C include

- *Candidates* can only have one *recruiter* and one *industry*,
- *Recruiters* don't link to *industries*

Depending on the queries and the objectives of the machine learning model, we could pick one schema or experiment with all three in the course of developing our application.

We decide to use the first schema, which will serve as a simple structure for our exploration, and a baseline structure for our experimentation.

To summarize:

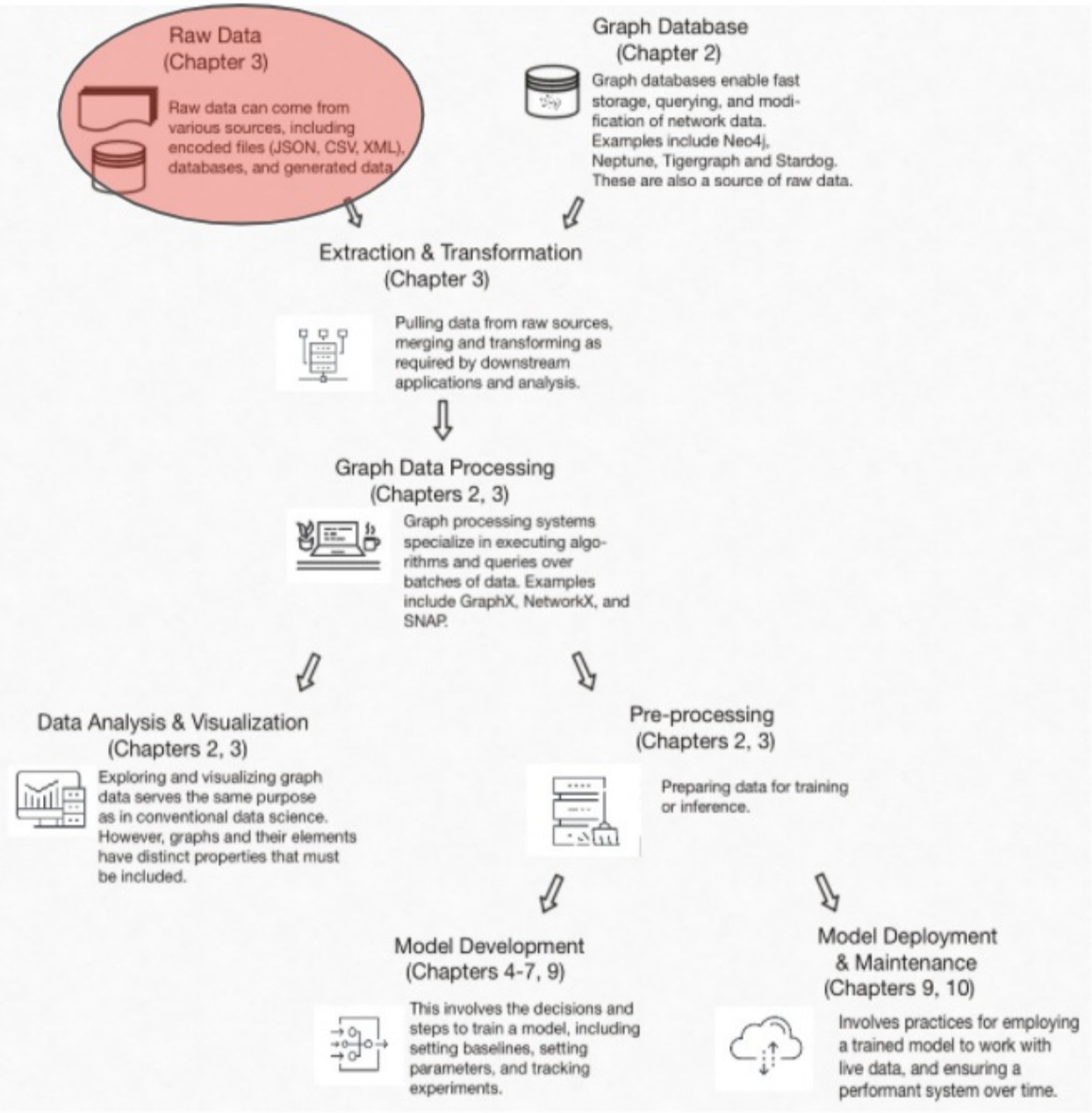
- **Data Model**. The data model we will use is the simple undirected graph, which consists of one node type (*candidate*) and one edge type (*knows*).
- **Conceptual Schema.** Our conceptual schema is visualized in figures 2.4, 2.5, and 2.6. No relational constraints exist.
- **Data resources** There are many freely available datasets for testing and experimenting with graph-based learning.
- **Heterogeneous/homogenous** graphs are those that have many types of nodes and edges (heterogenous) or only one type of node and edges (homogenous).

2.3 A data pipeline example

With the schema decided, let's walk through a simple example of a data pipeline from raw data. In Chapter 1, we summarized our workflow in Figure 1.13. In this section, we'll assume our objective is to create a simple data workflow that will take our data from a raw state, the recruiter data, perform light exploratory analysis, as in Section 2.1, and end with a preprocessed dataset that can be used by a GNN library, such as Pytorch Geometric. We'll be walking through each of these steps in detail but refer the reader to Chapter 1 if they want a refresher on the whole process.

In Chapters 3, we will return to this pipeline, and use its preprocessed data to create graph embeddings and to train GNN models.

2.3.1 Raw data



Raw data refers to data in its most primitive state; such data is the starting point for our pipeline. This data could be in various databases, serialized in some way, or generated.

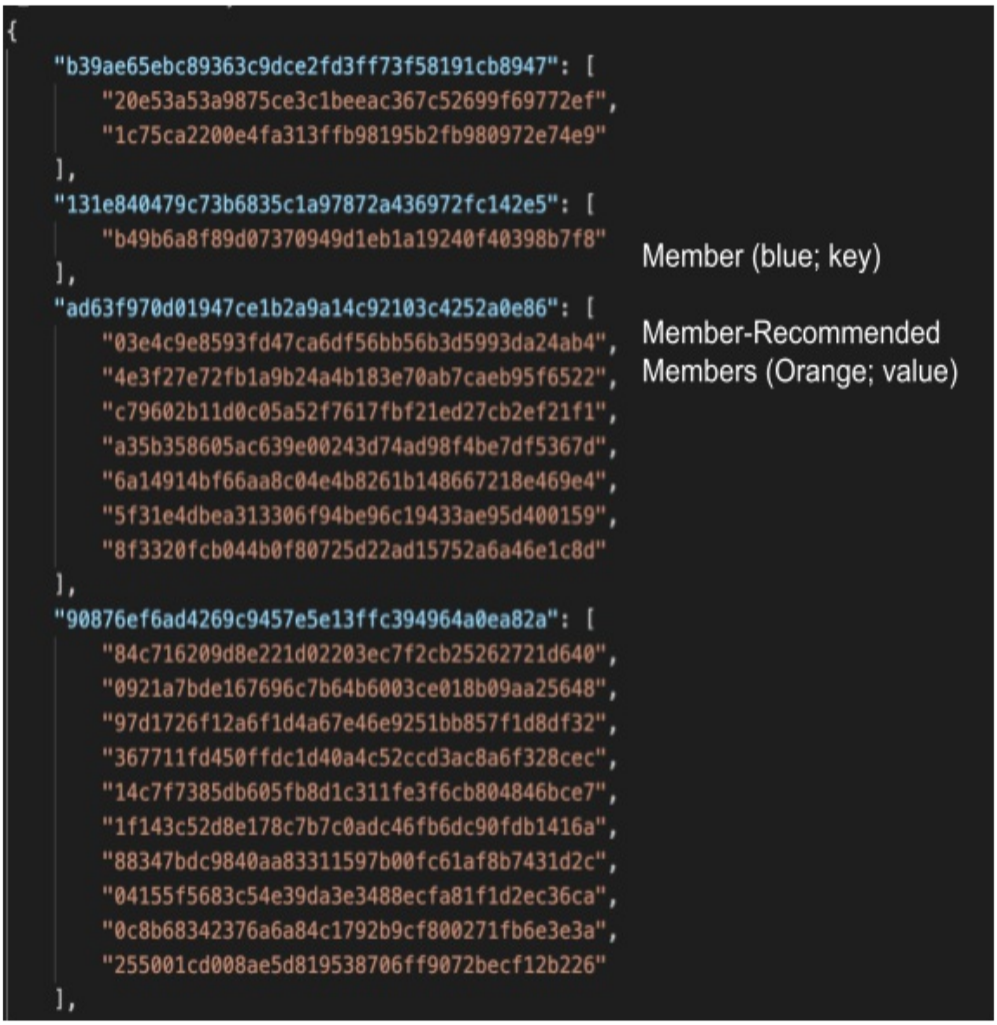
In the development stage of an application, it is important to know how closely the raw data used will match the live data used in production. One way to do this is by sampling from data archives.

As mentioned in section 2.1, there are at least two sources for our raw data, relational database tables that contain recommendation logs, and candidate profiles. To keep our example contained, we'll assume a friendly data engineer has queried the log data, and transformed it into a JSON format, where keys are a recommending candidate, and the values are the recommended candidates. From our profile data, we use two fields: *industry* and *job type*. For both data sources, our engineer has used a hash to protect personally identifiable information (PII). Thus, we can consider an alphanumeric hash the unique identifier of a candidate.

This data can be found here: https://github.com/keitabroadwater/gnns_in_action/tree/master/chapter_3.

For this chapter, we'll use JSON data, which assumes a recommendation implies a relationship. A snippet of this JSON data is shown in figure 2.7.

Figure 2.7 View of Raw Data: JSON File. This file is in key/value format (blue and orange, respectively). The keys are the members, and the values are their known relationships.



Data Encoding and Serialization

A consideration in the workflow is the choice of what data format is used in exporting and importing your data from one graph system to another. While in memory or a database, graph data is stored using that system's conventions. For transferring this data into another system, or sending it over the internet, encoding or serialization is used.

Before choosing an encoding format, one must have decided upon:

- Data Model - Simple model, property graph, or other?
- Schema - which entities in your data are nodes, edges, properties.
- Data Structure - These include the data structures discussed in Chapter

1, such as adjacency matrices
- Receiving systems - how does the receiving system (in our case GNN libraries and graph processing systems) accept data. What encodings and data structures are preferred. Is imported data automatically recognized, or is custom programming required to read in data.

Here are a few encoding choices you will encounter.

Language and system agnostic encodings formats: These are recommended when building a workflow as they allow the most flexibility in working amongst various systems and languages. However, how the data is arranged in these encodings may differ from system to system. So, an edge list in a csv file, with a set of headers, may not be accepted or interpreted in the same way between system 1 and system 2.

- JSON - Has advantages when reading from APIs, or feeding into javascript applications. Cytoscape.js, a graph visualization library, accepts data in JSON format.
- CSV - Accepted by many processing systems and databases. However, the required arrangement and labeling of the data differs from system to system.
- XML - GEXF (Graph Exchange XML Format) of course is an XML format.

Language Specific: Python, Java, and other languages have built-in encoding formats.

- Pickle - Python's format. Some systems accept Pickle encoded files. Despite this, unless your data pipeline or workflow is governed extensively by python, pickles should be used lightly. The same applies for other language-specific encodings.

System Driven: Specific software, systems, and libraries have their own encoding formats. Though these may be limited in usability between systems, and advantage is that the schema in such formats is consistent. Software and systems that have their own encoding format include SNAP, NetworkX, and Gephi.

Big Data: Aside from the language-agnostic formats used above, there are other encoding formats used for larger sizes of data.

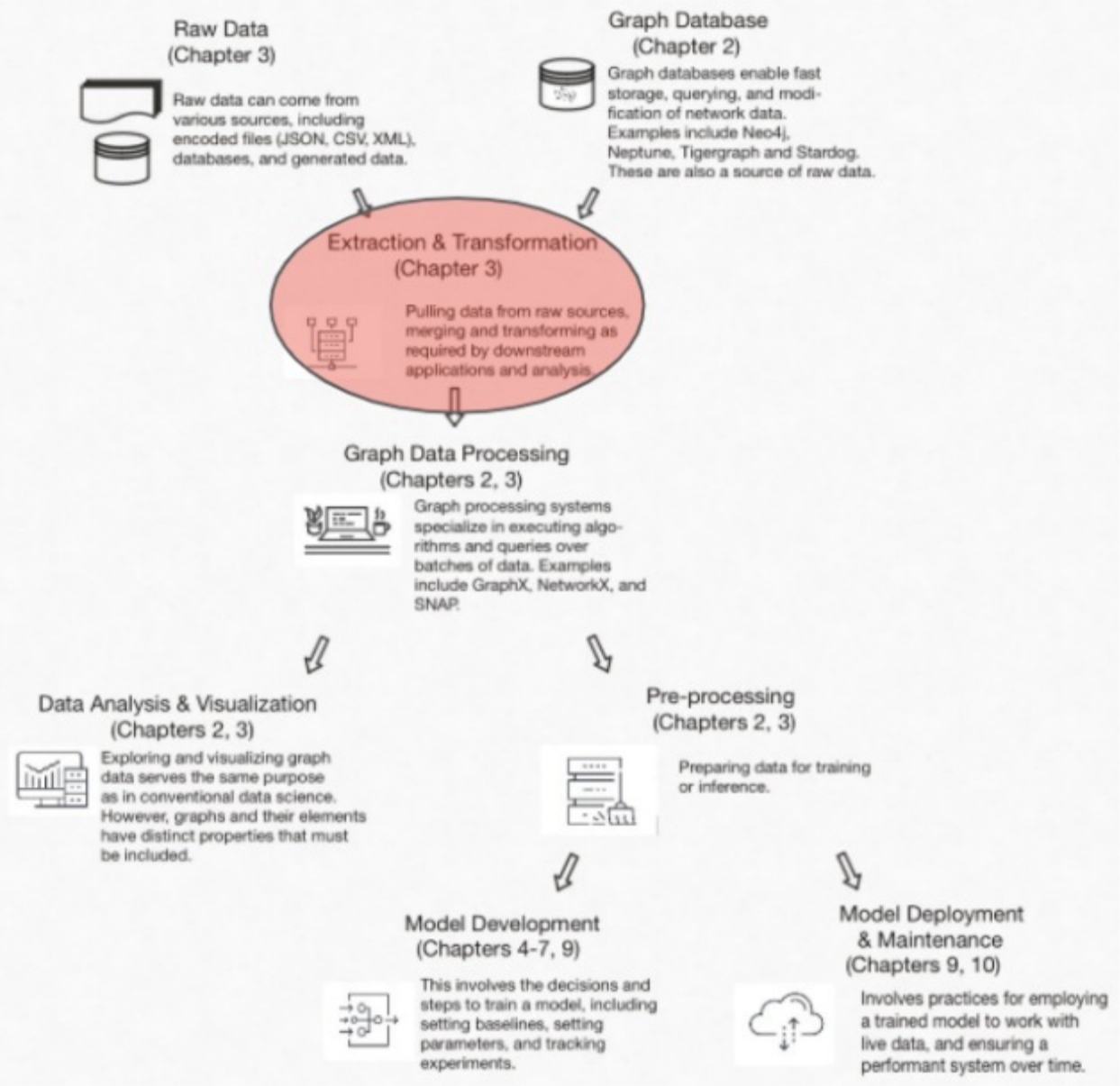
- Avro - This encoding is used extensively in Hadoop workflows

Matrix based. Since graphs can be expressed as matrix, there are a few formats that are based on this data structure. For sparse graphs, the following formats provide substantial memory savings and computational advantages (for lookups and matrix/vector multiplication):

- sparse column matrix (.csc filetype)
- sparse row matrix (.csr filetype)
- Matrix market format (.mtx filetype)

With these preliminaries out of the way, we'll present code that will fulfill the workflow.

2.3.2 ETL



With the schema chosen, and data sources established, the *extract/transform/load* step consists of taking raw data from its sources and through a series of steps, producing data that fits the schema and is ready for preprocessing or training.

For our data, this consists of programming a set of actions that begin with pulling the data from the various databases, and joining them as needed.

What we need is data that ends up in a specific format that we can input into a preprocessing step. This could be a JSON format, or an edge list. For either

of these examples (JSON and/or edge list), our schema is fulfilled; we have nodes (the individual persons), edges (the relationships between these people), with weights assumed to be 1 at this stage.

For our social graph example, we want to transform our raw data into a graph data structure, encoded in csv. This file can then be loaded into our graph processing system, NetworkX, and our GNN system, Pytorch Geometric. To summarize the next steps:

A Convert raw data file to edge list and adjacency matrix. Save as a csv file.

B Load into networkx for EDA & Visualization

C Load into Pytorch Geometric and preprocess.

Raw data to Adjacency Matrix and Edge List

Starting with our csv and json files, let's convert the data into two key data models we've learned: an edge list and an adjacency list, which we introduced in Chapter 1.

First using the *json* module, we place the data in the json file into a python dictionary:

Listing 2.1 Import JSON module. Import data from json file.

```
candidate_link_file = open('relationships_hashed.json',) #A
data = json.load(candidate_link_file) #B
candidate_link_file.close() #C
```

The python dictionary has the same structure as the json, with member hashes as keys, and their relationships as values.

Next, we create an adjacency list from this dictionary. This list will be stored as a text file. Each line of the file will contain the member hash, followed by hashes of that member's relationships.

This function is meant to illustrate the transformation of raw data into an

adjacency list. It was created for the social graph use case.

The inputs consist of:

- a dictionary of candidate referrals where the keys are members who have referred other candidates, and the values are lists of the people who were referred.
- a suffix to append to the file name

The outputs consist of:

- An encoded adjacency list in a txt file.
- A list of the node IDs found.

Listing 2.2 Function to create adjacency list from relationship dictionary.

```
def create_adjacency_list(data_dict, suffix=''):
   list_of_nodes = []

   for source_node in list(data_dict.keys()): #A

       if source_node not in list_of_nodes:
           list_of_nodes.append(source_node)

       for y in data_dict[source_node]: #B
           if y not in list_of_nodes: #B
               list_of_nodes.append(y) #B
           if y not in data_dict.keys(): #B
               data_dict[y]=[source_node] #B
           Else: #B
               if source_node not in data_dict[y]: #B
                   data_dict[y].append(source_node) #B
               else: continue #B

   g= open("adjacency_list_{}.txt".format(suffix),"w+") #C
   for source_node in list(data_dict.keys()): #D
       dt = ' '.join(data_dict[source_node]) #E
       print("{} {}".format(source_node, dt)) #F
       g.write("{} {} \n".format(source_node, dt)) #G

   g.close
   return list_of_nodes
```

Next, we create an edge list. As with the adjacency list, we transform the data to account for node pair symmetry.

As the adjacency list function in Listing 2.2, this function is meant to illustrate the transformation of raw data into an edge list. It was created for the social graph use case. This function has the same inputs as the previous function.

The outputs consist of:

- An edge adjacency list in a txt file
- Lists of the node IDs found and the edges generated

In Listing 2.3, we describe the code needed to create an edge list from a relationship dictionary.

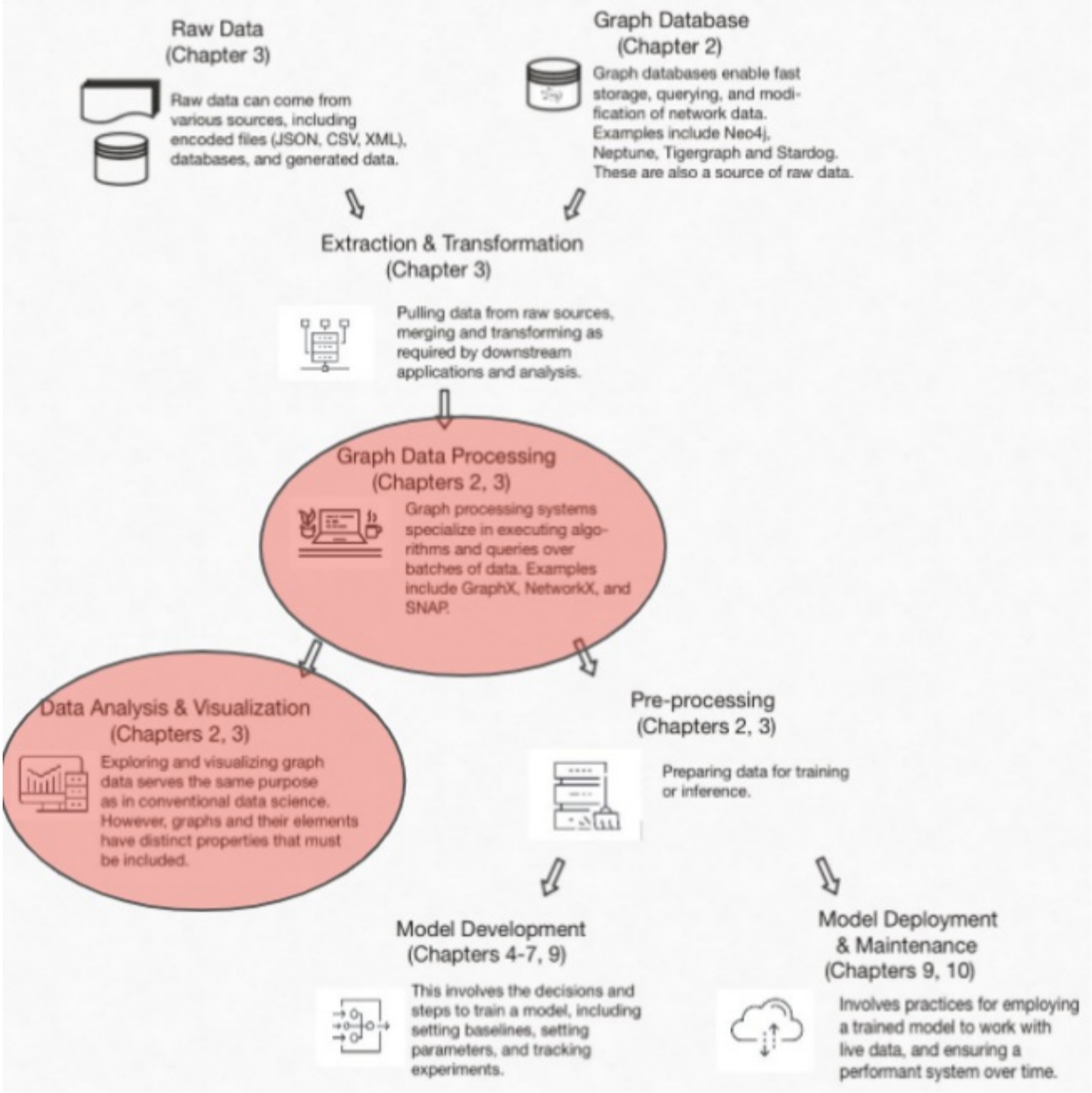
Listing 2.3 Function to create edge list from relationship dictionary.

```
def create_edge_list(data_dict, suffix=''):
   edge_list_file = open("edge_list_{}.txt".format(suffix),"w+")
   list_of_edges = []
   list_of_nodes_all = []

   for source_node in list(data_dict.keys()):
       if source_node not in list_of_nodes_all:
           list_of_nodes_all.append(source_node)
       list_of_connects = data_dict[source_node]

       for destination_node in list_of_connects: #A
           if destination_node not in list_of_nodes_all:
               list_of_nodes_all.append(destination_node)

           if {source_node, destination_node} not in list_of_edge
               print("{} {}".format(source_node, destination_node
               edge_list_file.write("{} {} \n".format(source_node
               list_of_edges.append({source_node, destination_nod

           else: continue

   edge_list_file.close
   return list_of_edges, list_of_nodes_all
```

In the next section and going forward, we will use the adjacency list to load

our graph into NetworkX. One thing to note about the differences between loading a graph using the adjacency list versus the edge list is that edge lists can't account for single, unlinked nodes. It turns out that quite a few of the candidates at Whole Staffing have not recommended anyone, and don't have edges associated with them. These nodes are invisible to an edge list representation of the data.

2.3.3 Data Exploration and Visualization



Next, we want to load our network data into a graph processing framework. We choose NetworkX, but as explained in chapter 2, there are several choices depending on your task and preferences. We choose NetworkX because we have a small graph, and want to do some light EDA and visualization.

With either our edge list or an adjacency list we just created, we can create a Networkx graph object by calling the `read_edgelist` or `read_adjlist` methods.

Next, we can load in the attributes *industry* and *job type*. In this example these attributes are loaded in as a dictionary, where the node IDs serve as keys.

With our graph loaded, we can explore our data, and inspect it to ensure it aligns with our assumptions. First, the count of nodes and edges should match our member count, and the number of edges created in our edge list, respectively.

Listing 2.4 Function to create edge list from relationship dictionary.

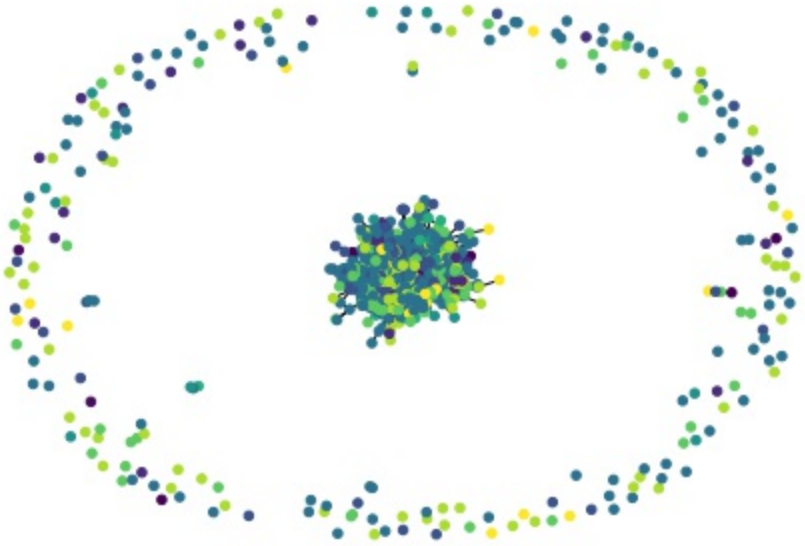
```
social_graph = nx.read_adjlist('adjacency_list_candidates.txt')
nx.set_node_attributes(social_graph, attribute_dict)
print(social_graph.number_of_nodes(), social_graph.number_of_edge
>> 1933 12239
```

We want to check how many connected components our graph has.

```
len(list((c for c in nx.connected_components(social_graph))))
>>> 219
```

The `connected_components` method generates the connected components of a graph; a visualization is shown in figure 2.8. There are hundreds of components, but when we inspect this data, we find that there is one large component of 1698 nodes, and the rest are composed of less than 4 nodes. Most of the disconnected components are singleton nodes (the candidates that never refer anyone).

Figure 2.8 The full graph, with its large connected component in the middle, surrounded by many smaller components. For our example, we will use only the nodes in the large connected component.



We are interested in this large connected component, and will work with that going forward. The `subgraph` method can help us to isolate this large component, as seen in Listing 5.

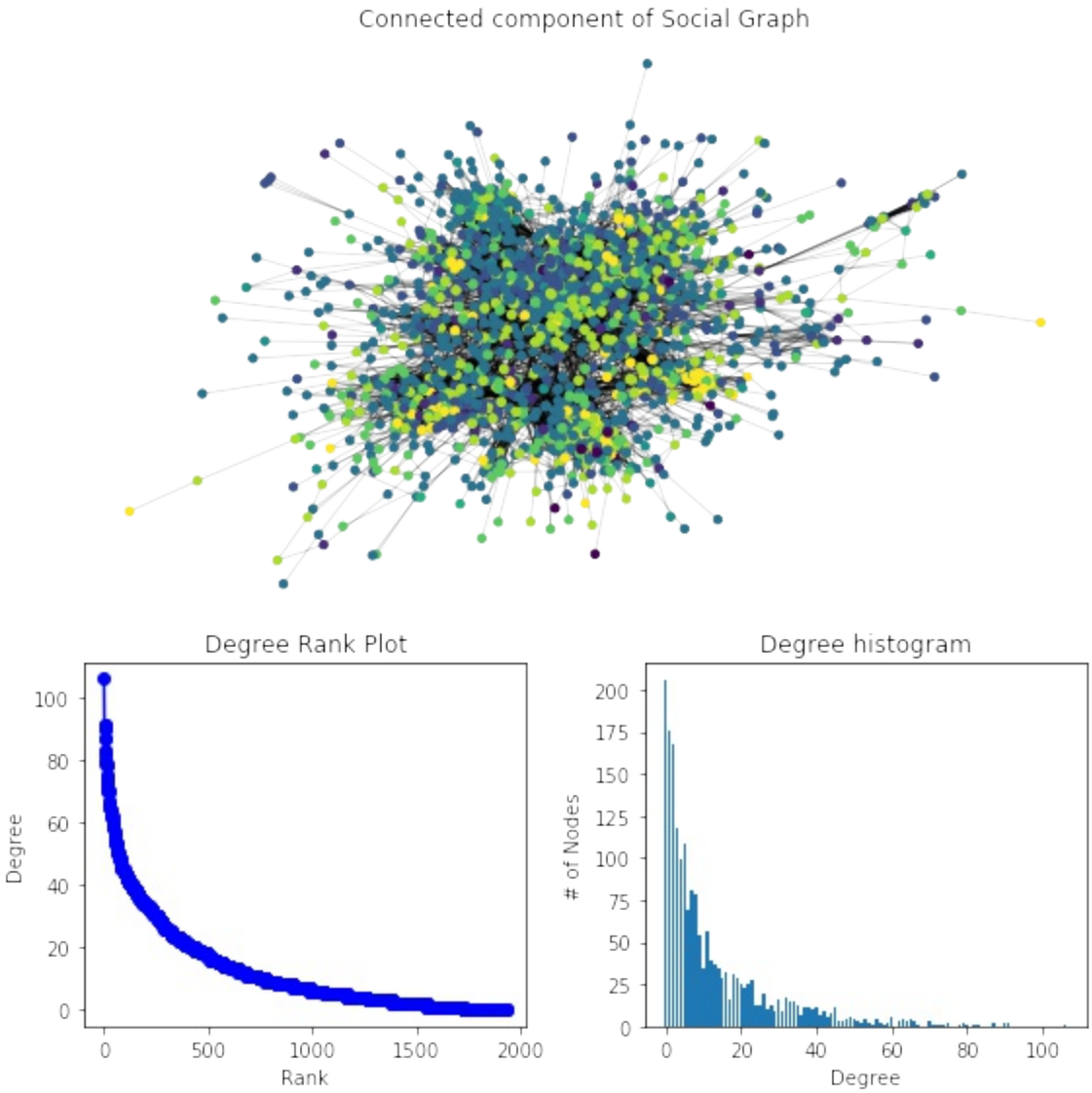
Lastly, we want NetworkX to give us statistics about our set of nodes. We also want to visualize our graph. For this, we'll use a recipe found in the NetworkX documentation.

Listing 2.5 Function visualize the social graph and show degree statistics.

```
## Modified from NetworkX documentation.

fig = plt.figure("Degree of a random graph", figsize=(8, 8))
# Create a gridspec for adding subplots of different sizes
axgrid = fig.add_gridspec(5, 4)

ax0 = fig.add_subplot(axgrid[0:3, :])

# 'Gcc' stands for 'graph connected component'
Gcc = social_graph.subgraph(sorted(nx.connected_components(social
pos = nx.spring_layout(Gcc, seed=10396953)  #B
nx.draw_networkx_nodes(Gcc, pos, ax=ax0, node_size=20) #C
nx.draw_networkx_edges(Gcc, pos, ax=ax0, alpha=0.4) #D
ax0.set_title("Connected component of Social Graph")
ax0.set_axis_off()

degree_sequence = sorted([d for n, d in social_graph.degree()], r
```

```
ax1 = fig.add_subplot(axgrid[3:, :2])
ax1.plot(degree_sequence, "b-", marker="o") #F
ax1.set_title("Degree Rank Plot")
ax1.set_ylabel("Degree")
ax1.set_xlabel("Rank")

ax2 = fig.add_subplot(axgrid[3:, 2:])
ax2.bar(*np.unique(degree_sequence, return_counts=True)) #G
ax2.set_title("Degree histogram")
ax2.set_xlabel("Degree")
ax2.set_ylabel("# of Nodes")

fig.tight_layout()
plt.show()
```

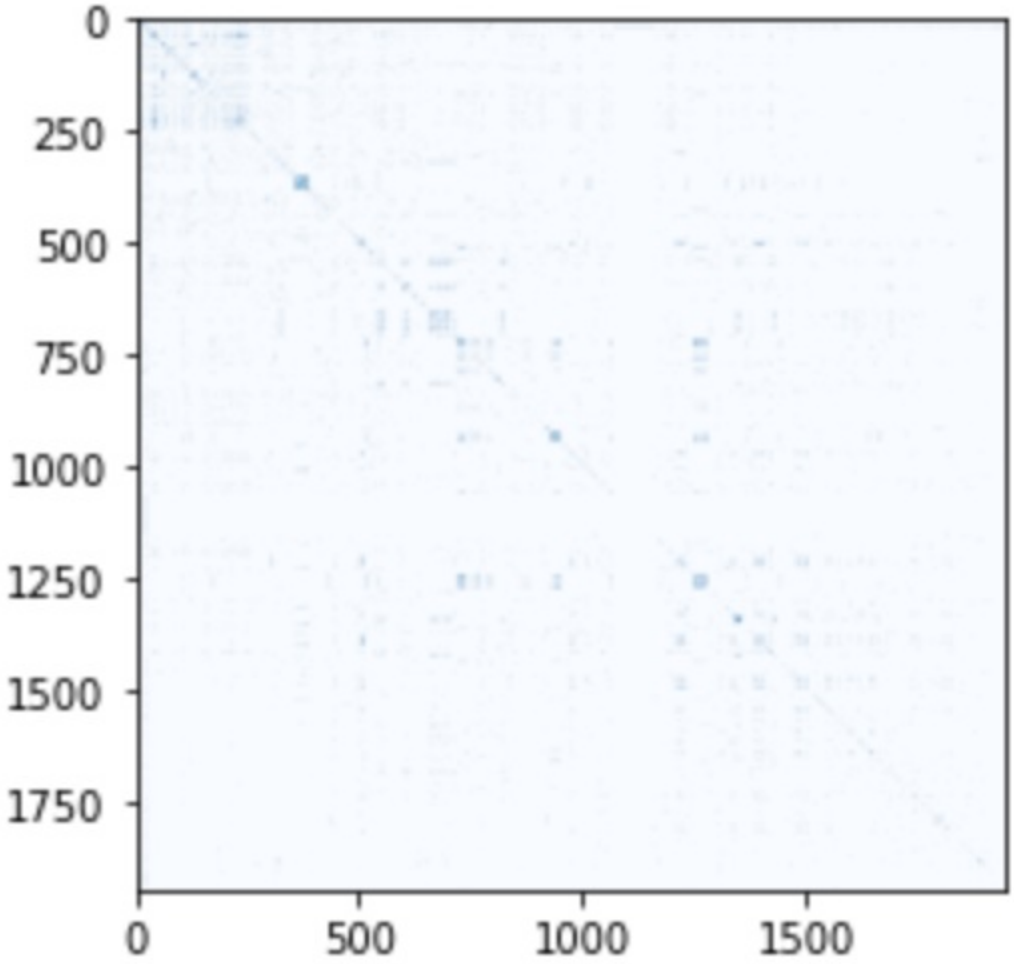
Figure 2.9 Visualization and statistics of the social graph and its large connected component. (Top) Network visualization using NetworkX default settings. (bottom left) a rank plot of node degree of the entire graph. We see that about 3/4ths of nodes have less than 20 adjacent nodes. (bottom right) A histogram of degree.



Lastly, we can visualize an adjacency matrix of our graph, shown in figure 2.10, using the below command:

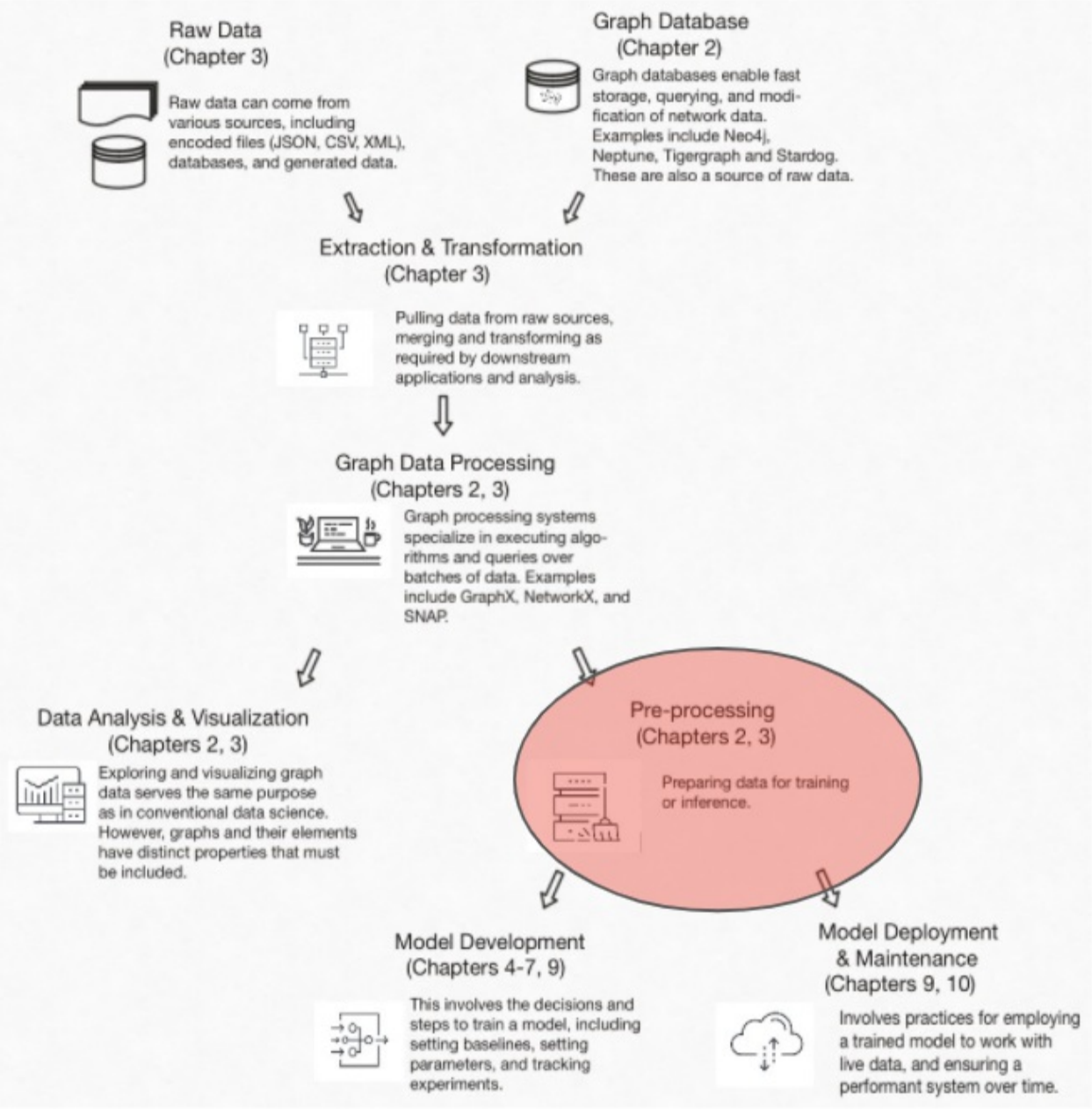
```
plt.imshow(nx.to_numpy_matrix(social_graph), aspect='equal',cmap=
```

Figure 2.10 A visualized adjacency matrix of our social graph. Vertical and horizontal values refer to respective nodes.



As with the numerical adjacency matrix, for our undirected graph, this visual adjacency matrix has symmetry down the diagonal.

2.3.4 Preprocessing: Pytorch Geometric



Preprocessing. For this book, preprocessing consists of adding properties, labels, or other metadata for use in downstream training or inference. For graph data, often we use graph algorithms to calculate the properties of nodes, edges, or sub-graphs.

An example for nodes is betweenness centrality. If our schema allows, we can calculate and attach such properties to the node entities of our data. To perform this, we'd take the output of the ETL step, say an edge list, and

import this into a graph processing framework to calculate betweenness centrality for each node. One this quantity is obtained, we can store it using a dictionary with the node ID as keys.

Betweenness Centrality

Betweenness Centrality is a measure of the tendency of a node to lie in the shortest paths from source to destination nodes. Given a graph with n nodes. You could determine the shortest path between every unique pair of nodes in this graph. We could take this set of shortest paths and look for the presence of a particular node. If the node appears in all or most of these paths, it has a high betweenness centrality, and would be considered to be highly influential. Conversely, if the node appears few times (or only once) in the set of shortest paths, it would have a low betweenness centrality, and a low influence.

Loading into GNN Environment. The last step we want to cover is inputting our preprocessed data into our GNN framework of choice. Both Pytorch Geometric and DGL have mechanisms to import custom data into their frameworks. Such methods allow for:

- Import from a graph library. For example, both PG and DGL support importing Networkx graph objects.
- Import of custom data in graph data structure. Edge list and adjacency lists can be directly imported.

Pytorch Geometric has a *Data* class which holds graph data, while DGL has a *DGLGraph* class.

After ETL and exploratory analysis, we are ready to load our graph into our GNN framework. For most of this book, we will use Pytorch Geometric (PyG) as our framework. For this section, we will focus on three modules within Pytorch Geometric:

- **Data** Module (torch_geometric.data): allows inspection, manipulation, and creation of data objects that are used by the pytorch geometric environment.
- **Utils** Module(torch_geometric.utils): Many useful methods. Helpful in

this section are methods that allow the quick import and export of graph data.
- **Datasets** Module(torch_geometric.datasets): Preloaded datasets, including benchmark datasets, and datasets from influential papers in the field.

Let's begin with the **Datasets** module. This module contains datasets that have already been preprocessed, and can readily be used by PyG's algorithms. When starting with PyG, having these datasets on hand allows experimentation with the algorithms without worrying about creating a data pipeline. As important, by studying the codebase underlying these datasets, we can glean clues on how to create our own custom datasets, as with our social graph.

That said, let's continue our exercise with our social graph. At the end of the last section, we had converted raw data into standard formats, and loaded our graph into a graph processing framework. Now, we want to load our data into the PyG environment, from where we can apply the respective algorithms.

Preprocessing in PyG has a few objectives:

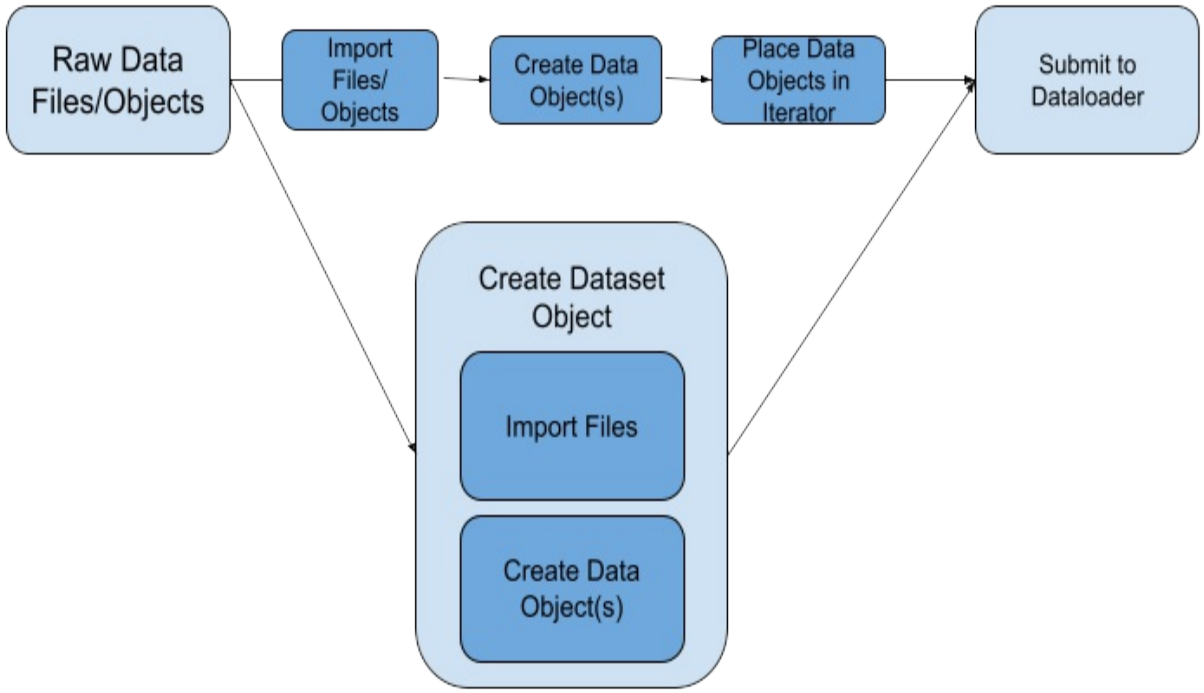
- Creating data objects with multiple attributes from the level of nodes and edges, to the subgraph and graph level
- Combining different data sources into one object or set of related objects
- Convert data into objects that can be processed using GPUs
- Allow splitting of training/testing/validation data
- Enable batching of data for training

These objectives are fulfilled by a hierarchy of classes within the **Data** module:

- **Data Class**: Creates graph objects. These objects can have optional built-in and custom-made attributes.
- **Dataset and InMemoryDataset Classes**: Basically creates a repeatable data preprocessing pipeline. You can start from raw data files, and add custom filters and transformations to achieve your preprocessed *data* objects. *Dataset* objects are larger than memory, while *InMemoryDataset* objects fit in memory.

- **Dataloader Class**: Batches data objects for model training.

Figure 2.11 Steps to preprocess data in Pytorch Geometric. From raw files, there are essentially two paths to prep data for ingestion by a PyG algorithm. The first path, shown above, directly creates an iterator of data instances, which is used by the dataloader. The second path mimics the first, but performs this process within the dataloader class.



As shown above, there are two paths to preprocess data that can be loaded into a training algorithm, one uses dataset class and the other goes without it. The advantages of using the dataset class is that it allows one to save not only the generated datasets, but preserve filtering and transformation details. Dataset objects have the flexibility to be modified to output variations of a dataset. On the other hand, if your custom dataset is simple or generated on the fly, and you have no use for saving the data or process long term,

bypassing dataset objects may serve you well.

So, in summary, when to use these different data-related classes:

- **Datasets Objects** - Pre-processed datasets for benchmarking or testing an algorithm or architecture. (not to be confused with Dataset (no ‘s’ at the end) objects)
- **Data Objects into Iterator** - Graph objects that are generated on the fly or for whom there is no need to save.
- **Dataset Object** - For graph objects that should be preserved, including the data pipeline, filtering and transformations, input raw data files and output processed data files. (not to be confused with Datasets (with ‘s’ at the end) objects)

With those basics, let’s preprocess our social graph data. We’ll cover the following cases:

a. Convert into *data* instance using NetworkX

b. Convert into data instance using input files

c. Convert to *dataset* instance

d. Convert *data* objects for use in *dataloader* without the *dataset* class

First, we’ll import the needed modules from PyG:

Listing 2.6 Required imports for this section, covering data object creation.

```
import torch
from torch_geometric.data import Data
from torch_geometric.data import InMemoryDataset
from torch_geometric import utils
```

Case A: Create PyG *data* object using *NetworkX* object

In the last sections, we have explored a graph expressed as a NetworkX *graph* object. PyG’s *util* module has a method that can directly create a PyG *data* object from a NetworkX *graph* object:

```
data = utils.from_networkx(social_graph)
```

The from_networkx method preserves nodes, edges and their attributes, but should be checked to ensure the translation from one module to another went smoothly.

Case B: Create PyG *data* object using *raw files*

To have more control over the import of data into PyG, we can begin with raw files, or files from any step in the ETL process. In our social graph case, we can begin with the edge list file created earlier.

Listing 2.7 Import the social graph into PyG starting with an edge file.

```
social_graph = nx.read_edgelist('edge_list2.txt')  #A

list_of_nodes = list(set(list(social_graph))) #B
indices_of_nodes = [list_of_nodes.index(x) for x in list_of_nodes

node_to_index = dict(zip(list_of_nodes, indices_of_nodes))  #D
index_to_node = dict(zip(indices_of_nodes, list_of_nodes))

list_edges = nx.convert.to_edgelist(social_graph)  #E
list_edges = list(list_edges)
named_edge_list_0 = [x[0] for x in list_edges] #F
named_edge_list_1 = [x[1] for x in list_edges]

indexed_edge_list_0 = [node_to_index[x] for x in named_edge_list_
indexed_edge_list_1 = [node_to_index[x] for x in named_edge_list_

x = torch.FloatTensor([[1] for x in range(len(list_of_nodes))])#
y = torch.FloatTensor([1]*974 + [0]*973) #I
y = y.long()

edge_index = torch.tensor([indexed_edge_list_0, indexed_edge_list

train_mask = torch.zeros(len(list_of_nodes), dtype=torch.uint8) #
train_mask[:int(0.8 * len(list_of_nodes))] = 1 #train only on the
test_mask = torch.zeros(len(list_of_nodes), dtype=torch.uint8) #t
test_mask[- int(0.2 * len(list_of_nodes)):] = 1
train_mask = train_mask.bool()
test_mask = test_mask.bool()

data = Data(x=x, y=y, edge_index=edge_index, train_mask=train_mas
```

We have created a *data* object from an edgelist file. Such an object can be inspected with PyG commands, though the set of commands is limited compared to a graph processing library.

Such a *data* object can also be further prepared so that it can be accessed by a dataloader, which we will cover below.

Case C: Create PyG *dataset* object using custom class and input files

If the listing above is suitable for my purposes, and I plan to use it repeatedly to call up graph data, a preferable option would be to create a permanent class that would include methods for the needed data pipeline. This is what the *dataset* class does. And to use it will not take much more effort, since we have done the work above.

Slightly modifying the script in listing 7, we can directly use it to create a *dataset* object. In this example, we name our *dataset* "MyOwnDataset", and have it inherit from *InMemoryDataset*, since our social graph is small enough to sit in memory. As discussed above, for larger graphs, data can be accessed from disc by having the *dataset* object inherit from *Dataset* instead of *InMemoryDataset*.

Listing 2.8 Class to create a dataset object.

```
class MyOwnDataset(InMemoryDataset):
    def __init__(self, root, transform=None, pre_transform=None):
        super(MyOwnDataset, self).__init__(root, transform, pre_t
        self.data, self.slices = torch.load(self.processed_paths[

    @property
    def raw_file_names(self): #B
        return []

    @property
    def processed_file_names(self): #C
        return ['../test.dataset']

    def download(self): #D
        # Download to `self.raw_dir`.
        pass

    def process(self): #E
```

```
# Read data into `Data` list.
data_list = []

eg = nx.read_edgelist('edge_list2.txt')

list_of_nodes = list(set(list(eg)))
indices_of_nodes = [list_of_nodes.index(x) for x in list_

node_to_index = dict(zip(list_of_nodes, indices_of_nodes)
index_to_node = dict(zip(indices_of_nodes, list_of_nodes)

list_edges = nx.convert.to_edgelist(eg)
list_edges = list(list_edges)
named_edge_list_0 = [x[0] for x in list_edges]
named_edge_list_1 = [x[1] for x in list_edges]

indexed_edge_list_0 = [node_to_index[x] for x in named_ed
indexed_edge_list_1 = [node_to_index[x] for x in named_ed

x = torch.FloatTensor([[1] for x in range(len(list_of_nod
y = torch.FloatTensor([1]*974 + [0]*973)
y = y.long()

edge_index = torch.tensor([indexed_edge_list_0, indexed_e

train_mask = torch.zeros(len(list_of_nodes), dtype=torch.
train_mask[:int(0.8 * len(list_of_nodes))] = 1 #train onl
test_mask = torch.zeros(len(list_of_nodes), dtype=torch.u
test_mask[- int(0.2 * len(list_of_nodes)):] = 1

train_mask = train_mask.bool()
test_mask = test_mask.bool()

data_example = Data(x=x, y=y, edge_index=edge_index, trai

data_list.append(data_example) #F

data, slices = self.collate(data_list)  #F
torch.save((data, slices), self.processed_paths[0])  #G
```

Case D: Create PyG *data* objects for use in *dataloader* without use of a *dataset* object

Lastly, we explain how to bypass *dataset* object creation and have the *dataloader* work directly with your *data* object. In the PyG documentation there is a section that outlines how to do this:

Just as in regular PyTorch, you do not have to use datasets, e.g., when you want to create synthetic data on the fly without saving them explicitly to disk. In this case, simply pass a regular python list holding

torch_geometric.data.Data objects and pass them to `torch_geometric.data.DataLoader`:

```
from torch_geometric.data import Data, DataLoader

data_list = [Data(...), ..., Data(...)]
loader = DataLoader(data_list, batch_size=32)
```

2.4 Summary

- Planning for a graph learning project involves more steps than in traditional machine learning projects.
- One important additional step is creating the data model and schema for our data.
- There are many reference schemas and datasets which span industry verticals, use cases, and GNN task.
- There are many encoding and serialization options for keeping data in memory or in raw files.
- The data pipeline up to model training consists of taking raw data, transforming it, performing EDA, and preprocessing.
- Adjacency lists, edge lists and other graph data structures can be generated from raw data.
- Pytorch Geometric, and GNN libraries in general have several ways to pre-process and load training data.

2.5 References

Graph Data Models and Schemas

Panos Alexopoulos, Semantic Modeling for Data, O'Reilly Media, 2020. Chapters 1-5.

Dave Bechberger, Josh Perryman, Graph Databases in Action, Manning

Publications, 2020. Chapter 2.

Denise Gosnell and Matthias Broecheler, The Practitioners Guide to Graph Data, O'Reilly Media, 2020. Chapter .

Alessandro Nego, Graph Powered Machine Learning, Manning Publications, 2021. Chapter 2.

Entity-Relationship Methods for Schema Creation

Richard Barker, Case*Method: Entity Relationship Modelling, Addison-Wesley, 1990.

Jaroslav Pokorný. 2016. Conceptual and Database Modelling of Graph Databases. In Proceedings of the 20th International Database Engineering & Applications Symposium (IDEAS '16). Association for Computing Machinery, New York, NY, USA, 370–377.

Pokorný, Jaroslav & Kovačič, Jiří. (2017). Integrity constraints in graph databases. Procedia Computer Science. 109. 975-981. 10.1016/j.procs.2017.05.456.

3 Graph Embeddings

This chapter covers

- Understanding graph embeddings and their limitations
- Using transductive and inductive techniques to create node embeddings
- Taking the example dataset, introduced in Chapter 2 and the Appendix A, to creating node embeddings

In the previous chapter, we outlined the creation of a social graph created by a recruiting firm. Nodes are job candidates, and edges represent relationships between job candidates. We generated graph data from raw data, in the form of edge lists and adjacency lists. We then used that data in a graph processing framework (NetworkX) and a GNN library (Pytorch Geometric). The nodes in this data included the candidate's *ID*, *job type* (accountant, engineer, etc), and *industry* (banking, retail, tech, etc).

We're next going to discuss how to take this graph and perform *graph embeddings.* Graph embeddings low-dimensional representations that can be generated for entire graphs, sub-graphs, nodes, and edges. They are central to graph-based learning and can be generated in many different ways, including with graph algorithms, linear algebra methods, and GNNs.

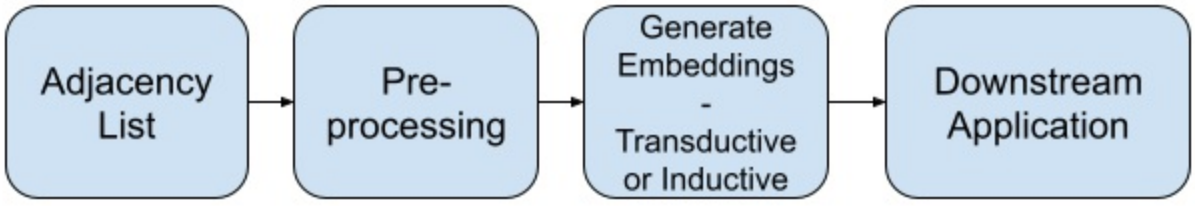
GNNs are special because embedding is inherent to their architectures. Previously when there was a need for a graph embedding in a machine learning application, the embedding and the model training were done with separate processes. With GNNs, embedding and model training are performed simultaneously when training the model.

This chapter covers graph representations, particularly node embeddings. We'll examine this space in general, and look at two ways of creating graph embeddings: using transductive and inductive methods. As part of this process, we'll create our first output from a GNN.

The objective for this chapter is to generate node embeddings from our graph.

As shown in Figure 3.1, we'll start with an adjacency list, do needed preprocessing, then use two techniques to create and visualize our node embeddings.

Figure 3.1 A high-level process for this chapter.



In the next part of the book, covering Chapters 4 to 6, we'll be focussing on the entire process of training GNN models, which are deeply connected to the concept of embedding.

For this chapter, first we'll do a deeper dive into embeddings as representations of graph entities. We'll examine and try transductive and inductive methods. In Section 3.2, we'll use Node2Vec, a transductive technique, then in Section 3.3, we'll meet our first graph neural network to generate the embeddings.

3.1 Graph Representations II

In Chapters 1 and 2, we touched on the notion of graph representations. We said then that a core sub-task of ML is to find ways to present data to our algorithms that allow them to learn from it. ML algorithms also output representations for use in downstream tasks. In this section we'll explore this concept more deeply and highlight its relevance to GNNs.

3.1.1 Overview of Embeddings

A **Data Representation** is a way of displaying, formatting, or showing data for some usage.

In language, a word can be written in different languages, different fonts.

That work can be written, spoken, embossed in Braille, or tapped out in Morse code.

Numbers can be written in Arabic, Roman, or Chinese characters. These numbers can be expressed in Binary, Hex, or Decimal notations. They can be written, spoken, or we can use our feet to count the number by stomping the ground. A given set of digital information can be shown or presented to software in many ways.

The usefulness of a representation is tied to the particular task of relevance. For our language example, effectively communicating a message requires a particular representation of words. If I want to order a pizza, writing out the order on paper won't help, even if all the details of the order are captured on paper. I could mail or fax the written order; but the fastest way is probably to speak the order over the phone.

A numeric example (from the *Deep Learning* textbook) is doing long division given two numbers. If the numbers are presented to the problem solver in any other way than arabic numerals, the solver will convert the numbers to the Arabic representation to solve the problem. So, whether the numbers are spoken, written out in Roman Numerals, or tapped out in morse code, before solving the division task the solver will convert the numbers to the arabic representation.

Bringing this back to graphs and machine learning, we covered a few ways to represent a graph such as using adjacency matrices and edge lists, in Chapter 1 and Appendix B. We also discussed high-level ways that graphs are represented, such as using arrays and hashes.

Adjacency matrices and edge lists are powerful enough to perform many graph analytical methods. Such data models enable network traversal on which many analytical methods are based. However, an adjacency matrix cannot convey more rich information about a network, including the features and properties of its elements, and more subtle topological information.

Table 3.1 Different methods for graph-representation

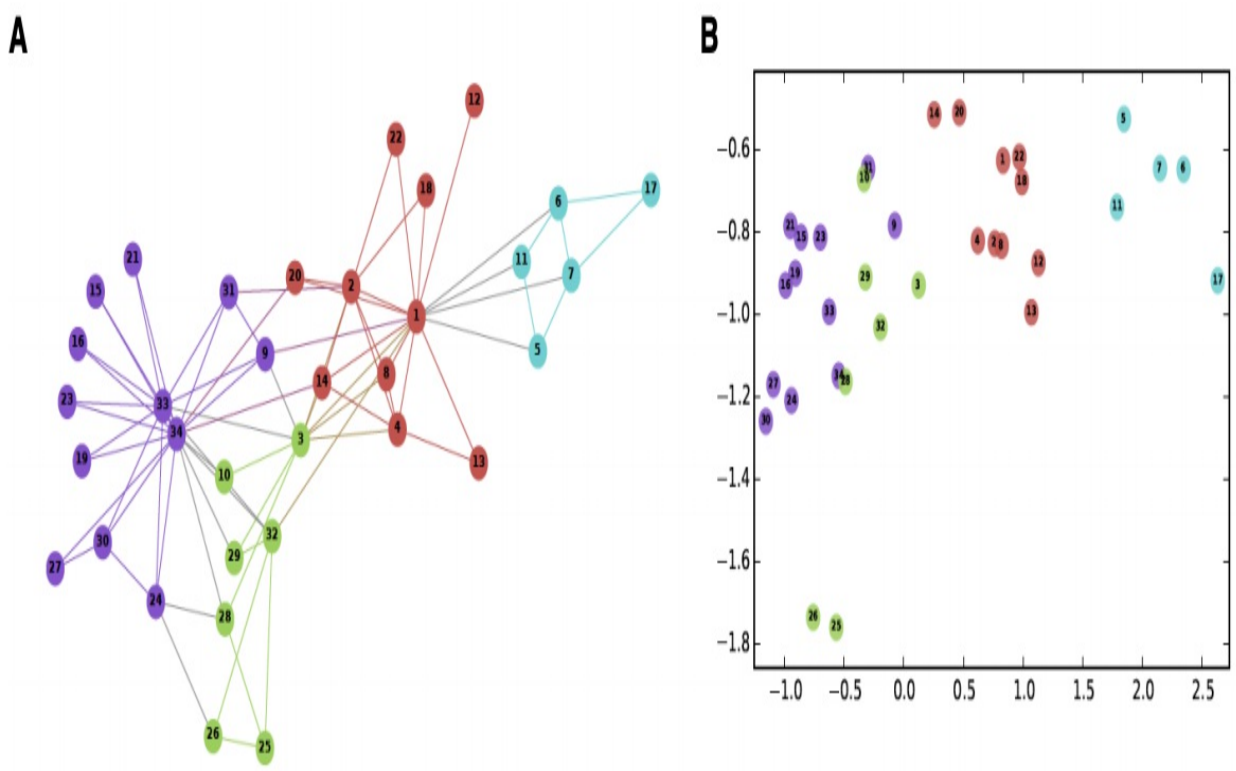
Representation	Description	Examples
Basic Data Representations	· Great for analytical methods that involve network traversal · Useful for some node classification algorithms · Information provided: Node and edge neighbors	· Adjacency list · Edge list · Adjacency matrix
Transductive (shallow) Embeddings	· Useless for data not trained on · Difficult to scale	· Deepwalk · Node2Vec · TransE · RESCAL · Graph Factorization · Spectral Techniques
Inductive Embeddings	· Models can be generalized to new and structurally different graphs · Represent	· GNNs can be used to inductively generate embeddings

To glean more information about a graph and its components, we turn to embedding techniques. An **embedding** is a low-dimensional numerical

representation of a graph, node, or edge that conveys information that can be used in multiple contexts.

For a graph, a node, or an edge, this numerical representation can be expressed in the form of a vector **x** that has **d** number of dimensions. This **vector** representation is called a **low-dimensional** representation of the entity if the dimensionality of the vector is lower than the initial dimensionality of the data.

Figure 3.2 A graph (left), and its two-dimensional representation (source: snap-stanford).



When reduced to two or three dimensions, vector representations allow us to create visualizations that can be used to inspect a graph. For example, in the figure above, we observe that nodes that are far apart in the graph, are far apart in the 2-D scatter plot.

The way an embedding is created determines the scope of its subsequent usage. Here we examine embedding methods that can be broadly classified as **transductive** and **inductive**.

Both methods have their trade-offs, and either can shine given the right problem.

Inductive learning techniques are equivalent to supervised learning. These learning methods refine a model using training data, verifies the model performance using test data, and is applied to newly observed data outside the training and test sets. The aim is to create a model which generalizes its training. As long as the new data follows the same distributional assumptions of the training and test data, there should be no need to retrain an inductive model.

Transductive learning techniques contrast with inductive learning in a few key ways:

- **Closed set of data for both training and prediction**. Transductive algorithms are not applied to new data outside of that used to train them.
- **Labeled and Unlabeled data are used in training**. Transductive algorithms make use of characteristics of unlabeled data such as similarities or geometric attributes to distinguish them.
- **The training directly outputs our predictions.** Unlike inductive models, there is no predictive model, only a set of predictions for our data points.
- **If we want to predict on new data, we must retrain our model.** Basically restating the first point.

What are the advantages of transductive models? The major advantage is that we can reduce the scope of the prediction problem. For induction, we are trying to create a generalized model that can be applied to any unseen data. For transduction, we are only concerned with the data we are presented with.

Disadvantages of transductive learning are that for many data points, this can be computationally costly. Also, our predictions can’t be applied to new data; in that case the model must be retrained.

In the context of node embedding, for a given graph, a transductive technique will directly produce embeddings. Thus, the result is essentially a lookup table. To make an embedding for a newly introduced node, it will be necessary to retrain using the complete set of nodes, including the new node.

Summary of Terms Related to Transductive Embedding Methods

Two additional terms related to transductive embedding methods and sometimes used interchangeably with it are **shallow embedding methods**, and **encoders**. Here, we will briefly distinguish these terms.

Transductive methods, explained above, are a large class of methods of which graph embedding is one application. So, outside of our present context of representation learnings, the attributes of transductive learning remain the same.

In machine learning, *shallow* is often used to refer to non-deep learning models or algorithms. Such models are distinguished from deep learning models in that they don't use multiple processing layers to produce an output from input data. In our context of graph/node embeddings, this term also refers to methods that are not deep learning based, but more specifically points to methods that mimic a simple lookup table, rather than a generalized model produced from a supervised learning algorithm.

Given the point of view that a shallow embedding is a lookup table, the 'model' that the transductive (or shallow) method produces is called an *encoder*. This encoder simply matches a given node (or graph) to its respective embedding in low dimensional space.

3.2 Transductive Embedding Technique: Node2Vec

Now we turn our attention to transductive techniques applied to graph embeddings. The goal of such methods is to deliver embeddings at the graph- or node-level that reflect the similarities, relationships, and structure of the given graphs. Since these are transductive methods, the resulting embeddings only apply to the given dataset, meaning they aren't valid for unseen nodes or graphs.

We'll focus on node embeddings. Random Walk methods are one type of transductive method for node embeddings. Of these, two well known methods are DeepWalk and Node2Vec. Such embedding methods borrow heavily from concepts in word embedding from NLP. Such methods embody

a few concepts:

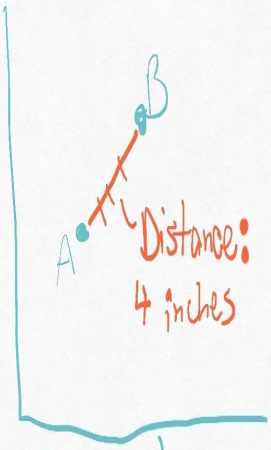
- Establishing **Node-to-Node Similarity**, or node context by performing **random walks**.
- **Optimizing using these similarities** to get embeddings that can predict of the members of a node's neighborhood

In the context of node embeddings, the characteristics and tradeoffs discussed here and in the previous section apply. We must also note an additional limitation: node embedding methods don't take into account node attributes or features. So, if there is rich data associated with our input nodes, methods like Node2Vec will ignore it.

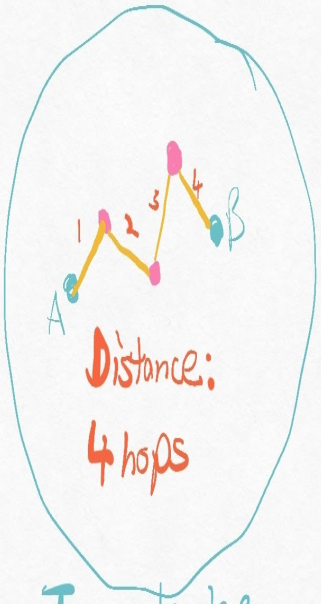
3.2.1 Node Similarity or Context

We want our output embeddings to provide information about the relative positions of the nodes and their respective neighborhoods. One way to convey this type of information is via the concept of **similarity**. In Euclidean domains, similarity often implies that entities have some geometric proximity, measured in terms of a distance and/or an angle. This is shown in Figure 3.3, where the distance is measured as the mean-squared metric distance between two points (think back to Pythagorus). For a graph, proximity or distance can be interpreted as how many edges one must traverse (or hop) to get from one node to another. So for graphs, developed notions of similarity between two nodes can hinge upon how close these nodes are in terms of number of hops. This is also shown in Figure 3.3, where we give an example of a 4 hop distance between nodes A and B.

Figure 3.3 Comparison of similarity concepts: (left) using distance on a plane, (right) using 'hops' along a graph.



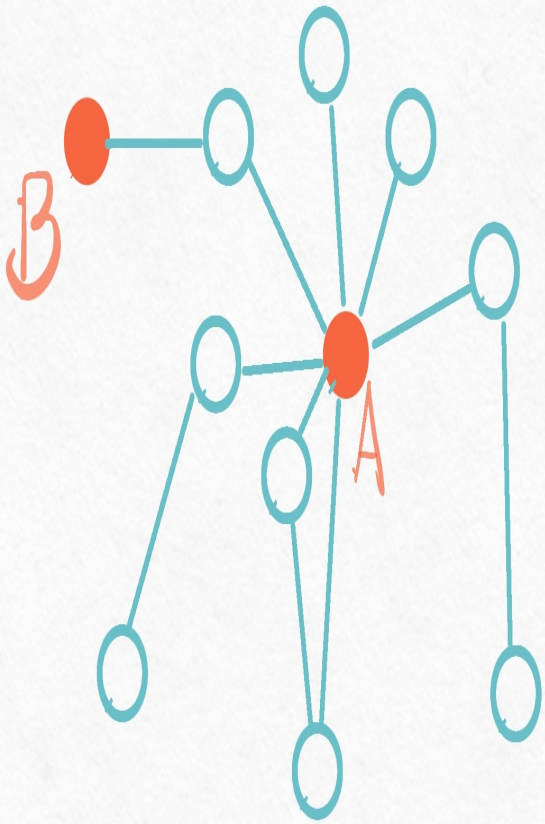
Two points on a plane.



Two nodes in a graph

Another way to think about proximity is in terms of probability: given two nodes (node A and node B), what is the chance that I will encounter node B if I start to hop from node A? In the figure, if the number of hops is 1, the probability is zero, given there is no way to reach node B from node A in one hop. If the number of hops is 2, and we add the restriction that no node can be encountered twice in a traversal, and the assumption that each direction is equally likely, the probability is 20%. We can see the steps needed to calculate this in Figure 3.4.

Figure 3.4 Illustrating the notion of proximity computed in terms of probability: given a walk from node A, the probability of encountering node B is a measure of proximity.


B
A

To make that calculation, I visually inspected the figure and used counting. However, in the real world, calculating such probability-based proximity on large and complex graphs would become rather intractable.

3.2.2 Random walks across graphs

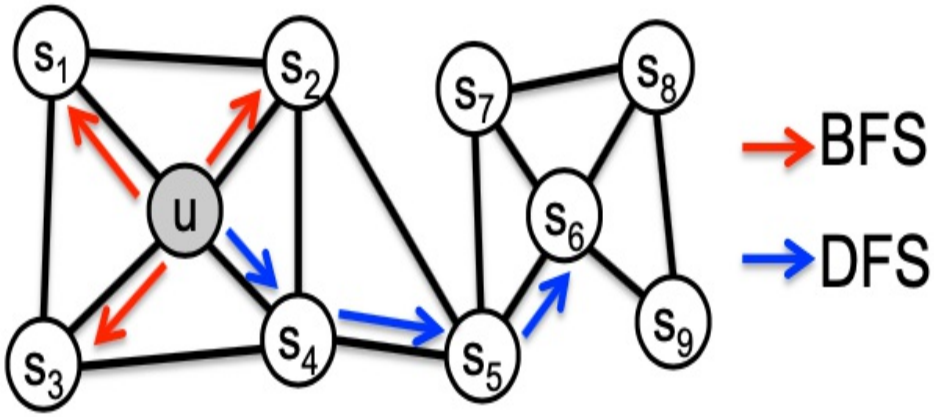
Random walk approaches expand on the ideas above by using random walks across the graph. With these, similarity between two nodes A and B is defined as the probability that one will encounter node B on a random graph traversal from node A. These walks are unrestricted in comparison with our simple example above, in that there is no restriction that prevents a walk from backtracking or encountering the same node multiple times.

Deepwalk determines similarity in this way by enacting several random walks of a fixed size for each node, and calculating similarities from these. In these walks, any path is equally likely to occur, making them unbiased.

Node2Vec improved on this by introducing tunable bias in these random walks. The idea is to be able to trade off learnings from a node's close-by neighborhood and from further away. Node2Vec captures this in two parameters:

- *p*: Tunes whether the path walked will return to the previous node.
- *q*: Tunes whether a DFS (depth first search, a hopping strategy that emphasizes faraway nodes) and BFS (breadth first search, a strategy that emphasizes nearby nodes). DFS and BFS are illustrated in Figure 3.5
- To mimic the DeepWalk algorithm, both *p* and *q* would be set to zero.

Figure 3.5 Illustration of DFS and BFS on a graph.



3.2.3 Optimization

As with all learning problems, Node2Vec has:

- An *objective function*. In this case, the log-probability of observing a node's neighborhood:

log(Pr(Ns(u)|f(u))

This log-probability is conditioned on the node's feature representation, f(u). This feature representation is the vector representation we want to end up with as an embedding.

- An *optimization target.* In this case, we wish to maximize the above objective function. In the Node2Vec paper, this is handled using stochastic gradient ascent. In the form above, for large graphs, computing the optimization is costly. So, a few simplifications are made based on assumptions of conditional independence and symmetry in a node's neighborhood.

3.2.4 Implementations and Uses of Node2Vec

There are a few implementations of Node2Vec out there. For our demonstrations, we will use the Node2Vec library at

https://github.com/eliorc/node2vec. As of this writing, it can be installed using:

```
pip install node2vec
```

Again, the aim here is to generate node embeddings from our social graph, then visualize them in 2D. For the visualization, we will use the T-SNE (pronounced tee snee) algorithm.

T-SNE

T-SNE, or T-distributed Stochastic Neighbor Embedding, is a dimensionality reduction technique. We will use it to reduce our node embedding vectors to two dimensions so that we can plot on a 2D figure.

TSNE and Node2Vec have somewhat similar goals: to present data in a low dimensional vector. The differences are in the starting point, and in the algorithms and assumptions. T-SNE starts with a vector and produces a vector, in 2 or 3 dimensions. Node2Vec and other graph embedding techniques start with a graph, and produce a vector where the resulting dimension is much lower than the dimensionality of the graph.

Let's start by loading the Node2Vec library and our social graph data. We'll generate node embeddings with 64 dimensions, then use T-SNE to further reduce these embeddings to 2 dimensions for plotting.

Listing 3.1 Read in our graph data, and import the Node2Vec library.

```
social_graph = nx.read_edgelist('edge_list2.txt')
social_graph.edges()
from node2vec import Node2Vec
```

Once the graph is loaded, we use create embeddings using Node2Vec.

Listing 3.2 Create embeddings using Node2Vec.

```
node2vec = Node2Vec(social_graph, dimensions=64, walk_length=30,
model = node2vec.fit(window=10, min_count=1, batch_words=4)  #B
model.wv.save_word2vec_format('EMBEDDING_FILENAME') #C
```

With a file with the embeddings, we can read in the file by line, as with a text file. Then transform these lines into a list of lists.

Listing 3.3 Read in the file of embeddings. Transform the data into a list of lists, with each sub-list a separate node embedding.

```
with open("EMBEDDING_FILENAME", "r") as social:
    lines = social.readlines()
embedded_lines = [x.split(' ')[1:] for x in lines[1:]] #A
n2v_embeddings = []
for line in embedded_lines:
    new_line = [float(y) for y in line] #B
    n2v_embeddings.append(new_line)
```

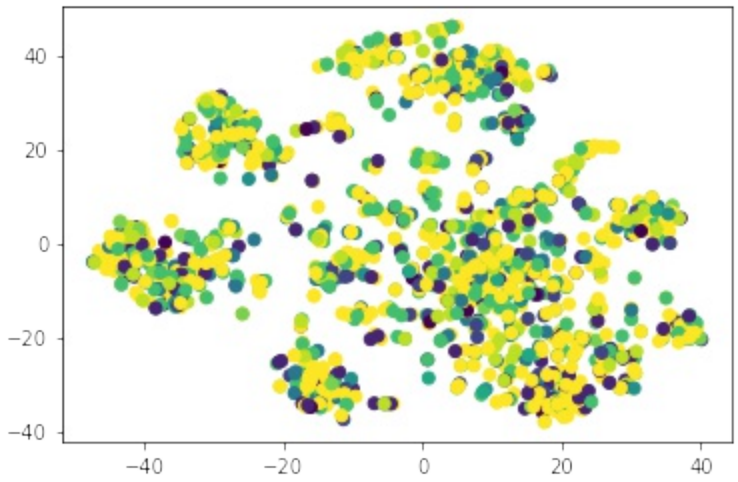
Finally, we fit a T-SNE model using our embeddings. We select the first and second features and plot.

Listing 3.4 Use T-SNE to Visualize Embeddings in 2-D.

```
tsne_model = TSNE(learning_rate=200)
tsne_features = tsne_model.fit_transform(n2v_embeddings)
# Select the 0th feature: xs
xs = tsne_features[:,0]
# Select the 1st feature: ys
ys = tsne_features[:,1]
plt.scatter(xs,ys)
plt.show()
```

Figure 3.6, is the visualization in 2 dimensions of the embeddings.

Figure 3.6 Embeddings of the social graph generated by Node2Vec. Colors correspond to *company_type*.



We observe structures and distinct clusters in our 2D representation. Next, we'll use a GNN with T-SNE to generate a similar plot.

3.3 Inductive Embedding Technique: GNN

In this section, we will briefly discuss a GNN used as an inductive method to generate node embeddings. We will then implement a simple GCN architecture using pytorch geometric. This GCN will generate the node embeddings for our social graph. Then in subsequent chapters, we'll deep dive into the GCN, and other important GNN architectures.

Graph Neural Networks can be thought of as end-to-end machine learning on graphs. This is because previous methods of graph-based machine learning were piecemeal, combining the results of several separate and distinct processes. Before the advent of GNN methods, to perform node classification, one would use a process where a node embedding technique was used (like Node2Vec), and its output was rolled as a feature into a separate machine learning method (like a random forest or linear regression).

With GNNs, the representation learning is inherent in the architecture. So, in this section, we will take advantage of this and directly work with the learned embeddings produced from our social graph.

(Let us note that while the embedding described here with a GNN is an inductive method, transductive methods have been used with GNNs for

downstream tasks [Rossi].)

3.3.1 Traits of Inductive Embeddings

In section 3.3.2, we outlined some of the characteristics and tradeoffs of transductive methods as applied to node embeddings. We list here some characteristics of inductive methods as a contrast:

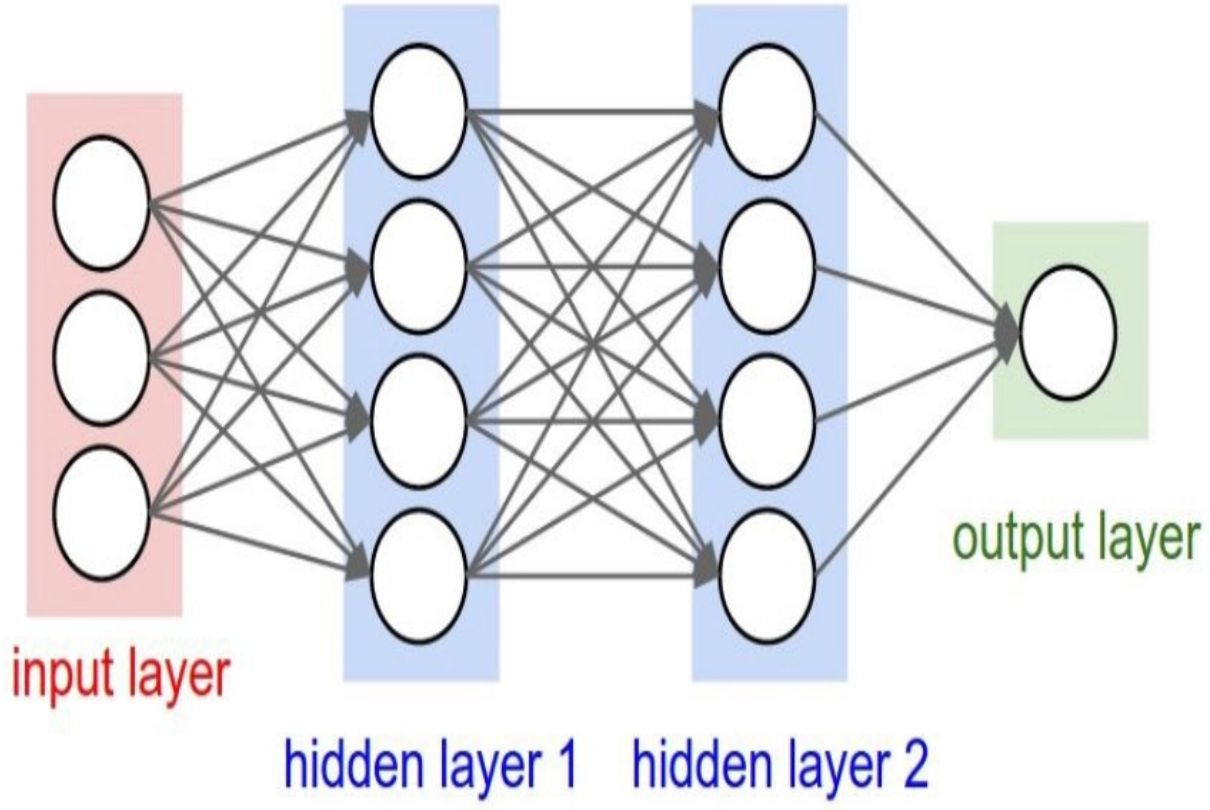
- **Node Features**: We learned that existing graph transductive methods don't take into account node attributes. Inductive methods can and do incorporate attributes and node features when calculating embeddings.
- **Deep vs Shallow**: We noted that transductive node embedding methods are akin to a lookup table. The lookup table can be seen as the transformation layer that converts a node to its respective vector embedding.
  For GNNs, our training algorithm results in fixing the parameters of a model.
- **Generalizable Model for Embeddings**: Our resulting model can be applied to unseen data, as opposed to the transductive way, which was limited to the training data.

Next, we take a high level look at GNNs and how we generate embeddings from them.

3.3.2 Deep Learning and GNNs

Deep Learning methods in general are composed of building blocks, or layers, that take some tensor-based input, and produce an output that is transformed as it flows through the various layers. At the end, more transformations and aggregations are applied to yield a prediction. However, often the output of the hidden layers are directly exploited for other tasks within the model architecture or are used as inputs to other models. A simple example, known as a multilayer perceptron or MLP, is shown in figure 3.7.

Figure 3.7 Layers in a multilayer perceptron.



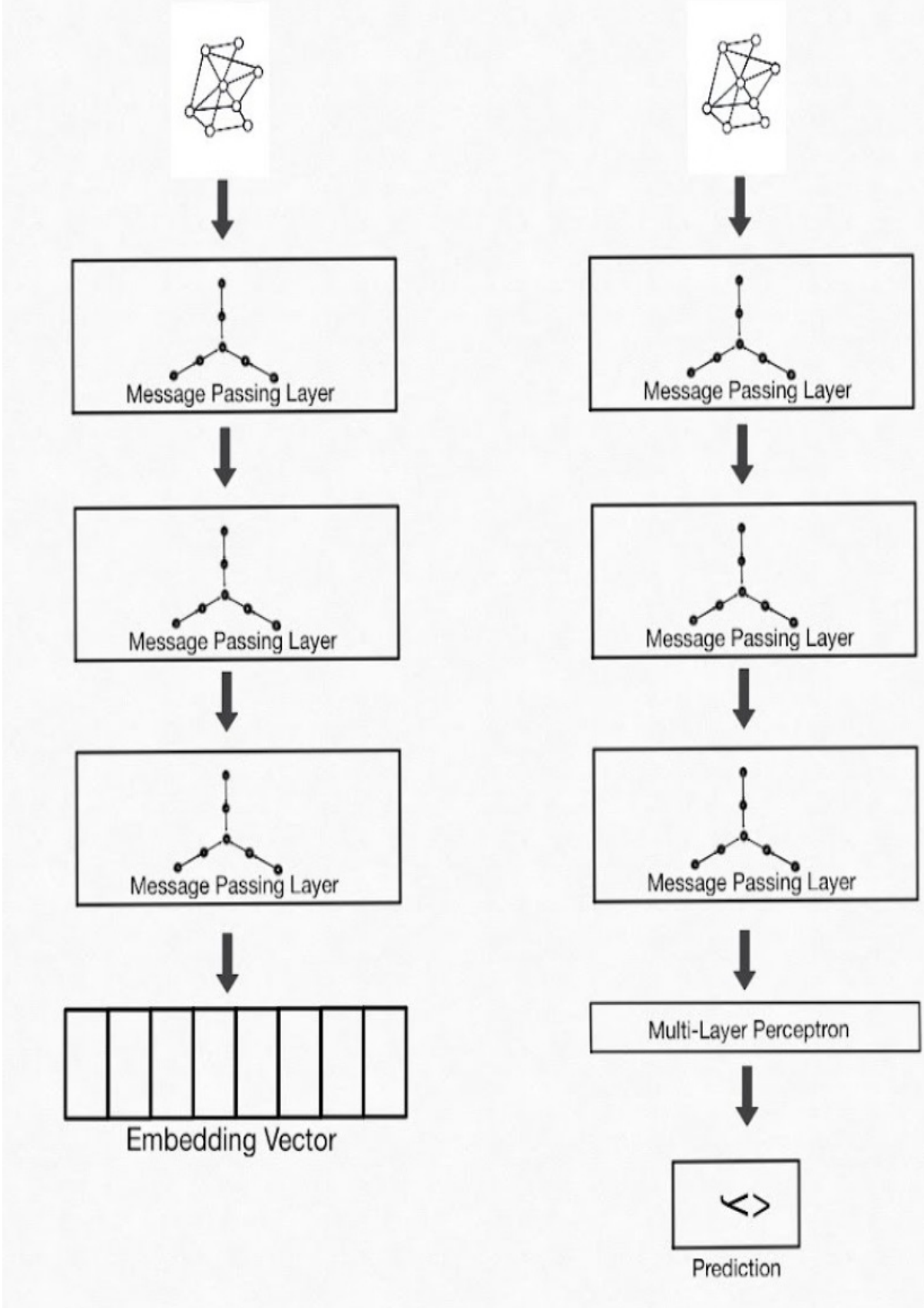
This same idea applies to graph neural networks. While the architecture of GNNs is starkly different in many ways from, say feed forward neural networks, there are some analogues. In many of the graph neural networks we will study, a graph in tensor form is passed into the GNN architecture, and one or more iterations of a process called **message passing** is applied. These message passing processes could be thought of as layers, as shown in figure 3.8.

For a feed-forward network, like the one in figure 3.7, information is passed between the nodes of our neural network. In a GNN, message passing or information passing occurs across the vertices of the graph. Over each message passing step, each vertex collects information from vertices one hop away. So, if we want our node representations to take account of nodes from 3 hops away from each node, we conceivably need 3 message passing layers. There are diminishing returns from adding message passing layers, which we'll explore in later chapters.

Different message passing schemes lead to different flavors of GNNs. So, for each GNN we study in this book, we'll pay close attention to the math and code implementation for message passing.

After message passing, the resulting tensor is passed through feed-forward layers that result in a prediction. In the top of figure 3.8, which illustrates a GNN model for node prediction, after the data flows through message passing layers, the tensor then passes through an additional multi-layer perceptron and activation function to output a prediction.

Figure 3.8 (top) A simple GNN architecture diagram. A graph is input on the left, encountering node-information-passing layers. This is followed by multi-layer perceptron layers. After an activation is applied, a prediction is yielded. (bottom) A graph neural network architecture, similar to the upper figure, but after passing through the message passing layers, instead of passing through MLP layers and an activation, the output data can be used as embeddings.


Message Passing Layer
Message Passing Layer
Message Passing Layer
Embedding Vector
Message Passing Layer
Message Passing Layer
Message Passing Layer
Multi-Layer Perceptron
Prediction

However, as with the feed-forward neural network illustrated above, we can extract the output of any of the hidden layers and work directly with that output. For GNNs, the output of hidden message passing layers are node embeddings.

3.3.3 Using Pytorch Geometric

Pytorch Geometric, like Pytorch, has several built in GNN layers, including for graph convolutional networks, the type we will use in our example. GCNs are often used for node classification tasks, which we will cover in chapter 4. For now, we will just extract the embedding from our last GCN layer, visualize them using T-SNE. From the first half of this chapter, we know how to create a pytorch dataset object, and will use that here.

3.3.4 Our Process/Pipeline

To generate node embeddings from our social graph, we:

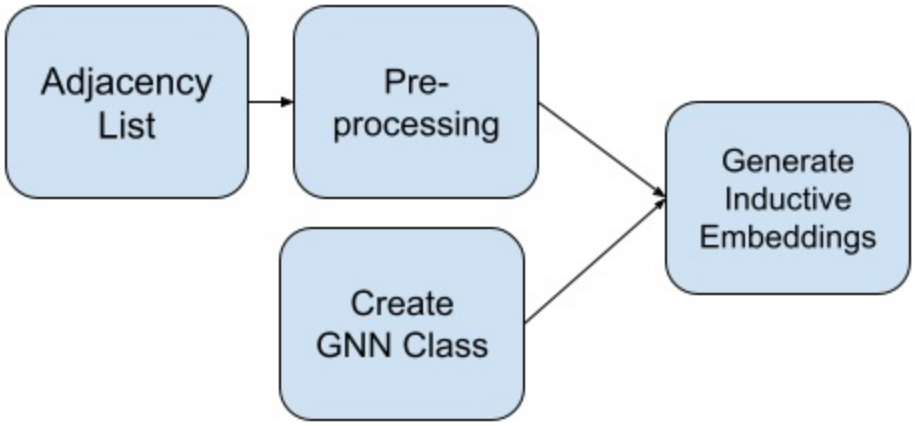
1. Begin with its adjacency list file,
2. Load it into a Pytorch Geometric dataset.
3. Initialize a GNN model.
4. Pass our data through this GNN to generate embeddings

For preprocessing, we will load the data into a pytorch geometric *dataset* object using the script we already created in section 3.2.

We will then create a simple architecture with three GCN layers (from PyG's built-in layers) to produce our embeddings. This process is shown in figure 3.9.

Unlike in later chapters, there is no need to train a model to make the embeddings. We could refine our embeddings against some criteria by doing training, which is exactly how GNNs work: the graph embeddings as well as the neural network parameters are optimized for prediction tasks.

Figure 3.9 Flowchart of this section's process.



Upon creating the dataset, we can inspect the graph's and dataset's properties.

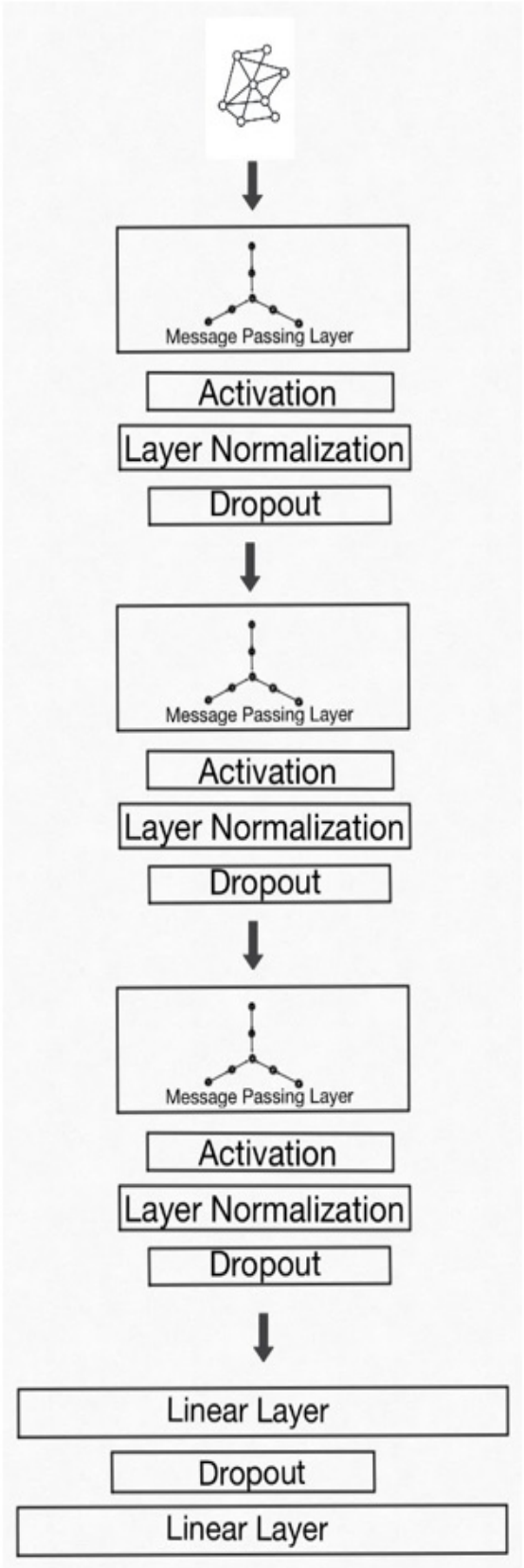
Listing 3.5 Commands to create dataset using code in listing 3.8, and to gather summary statistics about the graph.

```
dataset = MyOwnDataset('')
data = dataset[0]
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f
print(f'Contains isolated nodes: {data.contains_isolated_nodes()}
print(f'Contains self-loops: {data.contains_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')
Data(edge_index=[2, 12628], x=[1947, 1], y=[1947])
==========================================================
Number of nodes: 1947
Number of edges: 12628
Average node degree: 6.49
Contains isolated nodes: False
Contains self-loops: False
Is undirected: False
```

The GNN architecture for our example

Let's take a closer look at the layers in our example architecture.

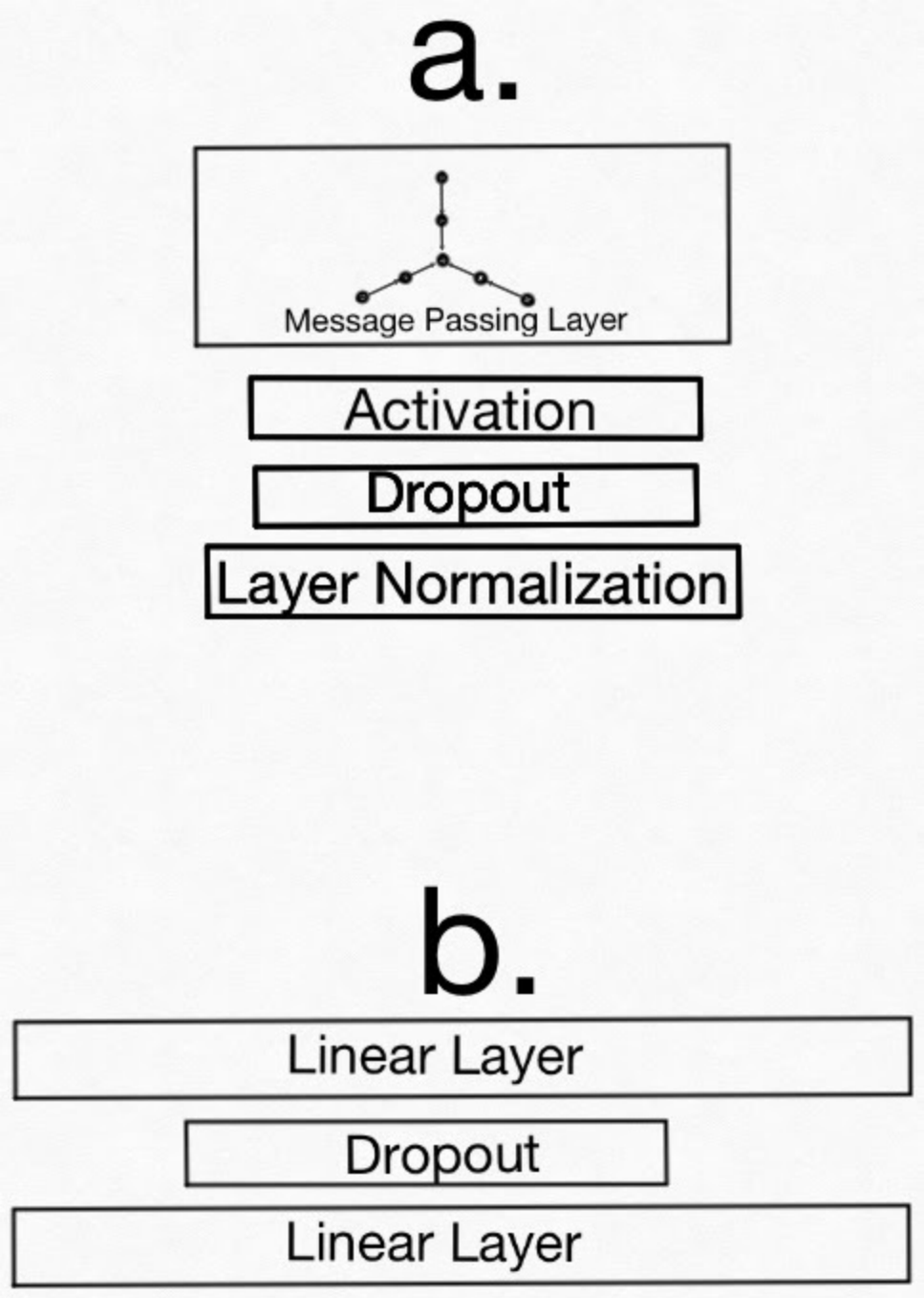
Figure 3.10 Architecture for this example. At the start, we have an input graph. This data is passed through 3 GNN layers, then through an MLP. Each time the graph data passes through one of the GNN layers, the output is an embedding.


Message Passing Layer
Activation
Layer Normalization
Dropout
Message Passing Layer
Activation
Layer Normalization
Dropout
Message Passing Layer
Activation
Layer Normalization
Dropout
Linear Layer
Dropout
Linear Layer

The GNN layers consist of:

- **A Message Passing layer**: where information gets passed between vertices
- **An Activation layer**: where a non-linear function is applied to the previous output
- **A Dropout layer**: switching off some of the output units.
- **A Layer Normalization laye**r: a normalization of the activated output to a mean of zero and a variance of 1.

Figure 3.11 (top) Diagram of GNN layers: Message Passing layer, Activation, Dropout, and Layer Normalization. (bottom) Diagram of our Feed Forward layers: Linear Layer, Dropout, and a second Linear Layer.



The feed-forward layers consist of:

- A Linear layer: A classic linear function, in the form of a single line of input neural nodes.
- A Dropout Layer
- A second Linear Layer

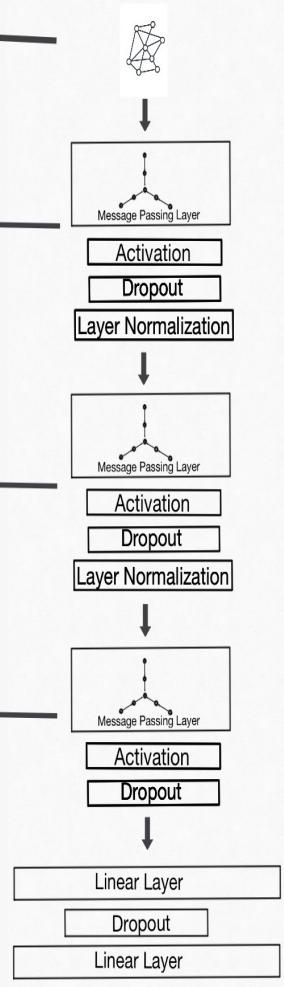
In pytorch, a way to create neural network architectures is by creating a class which inherits from the *torch.nn.Module* class. Listing 4.6 shows the our class, consisting of:

- An *__init__* method, which defines and initializes parameter weights for our layers.
- A *forward* method, which governs how data passes forward through our architecture.

In addition, there are a few hyper-parameters related to the input and output sizes of the message passing, layer norm, and linear layers, and the dropout.

Figure 3.12 The *__init__* and *forward* methods of our GNN class. This architecture contains 3 GCN layers, and a set of MLP layers. On the right, the diagram from Figure 3.10 is mapped to the code to make clear each individual step.

```python
def __init__(self):
    super(SimpleGNN, self).__init__()
    self.conv1 = GCNConv(datasetd.num_node_features, 16)
    self.conv2 = GCNConv(16, 16)
    self.conv3 = GCNConv(16, 16)
    self.lns = nn.LayerNorm(16)
    self.post_mp = nn.Sequential(
        nn.Linear(16, 16), nn.Dropout(0.25),
        nn.Linear(16, 2))

def forward(self, data):
    x, edge_index = data.x, data.edge_index

    x = self.conv1(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p= .25,training=self.training)
    x = self.lns(x)

    x = self.conv2(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p= .25,training=self.training)
    x = self.lns(x)

    x = self.conv3(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, p= .25,training=self.training)

    x = self.post_mp(x)

    return x ,F.log_softmax(x, dim=1)
```



Node Embeddings

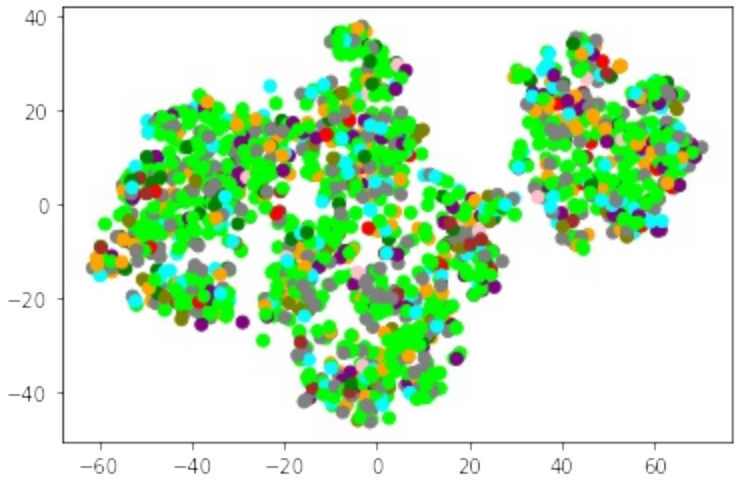
We've created our dataset and our GNN architecture. In Listing 3.7, we pass the graph data through our architecture and visualize the resulting embeddings. There is no model training involved, only a pass through, which consists of two lines of code. The first line initializes our model and its weights. The second line passes our graph data through this model.

Listing 3.7 Code to pass graph data through the GNN and plot the results.

```
model = SimpleGNN() #A

embedding, out = model(data) #B

color_list = ["red", "orange", "green", "blue", "purple", "brown"
colors = [] #D
colors += [color_list[y] for y in data.y.detach().numpy()] #E

xs, ys = zip(*TSNE().fit_transform(embedding.detach().numpy())) #
plt.scatter(xs, ys, c=colors) #G
```

Finally, we have the visualization of the node embeddings. We see clusters, but no correlation between company type and the clusters. By adjusting the GNN layers, the hyper-parameters, or the graph data (we have given every node a unit feature in this example), we can adjust the embeddings output. Ultimately, for a given application, the performance of the application will determine the efficacy of the embeddings.

Figure 3.13 Visualization of embeddings generated from GNN. Colors correspond to *company_type*.



3.4 Summary

- Node and Graph embeddings are powerful methods to extract insights from our data, and can serve as inputs/features in our machine learning models. There are several independent methods for generating such embeddings. Graph Neural Networks have embedding built into the architecture.
- For producing embeddings methods can be classed as inductive or transductive. Inductive learning methods are akin to supervised learning. GNNs are inductive methods. Transductive methods learn and ‘predicts’ on a closed dataset, train on labeled and unlabeled data, and basically acts as a lookup table rather than a predictive model. Random walk methods like Node2Vec are transductive.
- Node Embeddings can directly be used as features in traditional machine learning models, whether from transductive methods or from the output of a GNN.
- Embeddings are low-dimensional representations of graph objects. By using dimensionality reduction techniques like T-SNE to further reduce embeddings to two dimensions, we can visualize our graphs, and draw further insights by inspection.

3.5 References

Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for

networks." Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. 2016.

Duong, Chi Thang, et al. "On node features for graph neural networks." arXiv preprint arXiv:1911.08795 (2019).

Perozzi, Bryan, Rami Al-Rfou, and Steven Skiena. "Deepwalk: Online learning of social representations." Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining. 2014.

Hamilton, William L. "Graph representation learning." Synthesis Lectures on Artificial Intelligence and Machine Learning 14.3 (2020): 1-159.

Rossi, A., Tiezzi, M., Dimitri, G.M., Bianchini, M., Maggini, M., & Scarselli, F. (2018).

"Inductive–Transductive Learning with Graph Neural Networks",

In Artificial Neural Networks in Pattern Recognition (pp.201-212).

Berlin: Springer-Verlag.

4 Graph Convolutional Networks (GCNs) & GraphSage

This chapter covers

- Introducing GraphSage and GCN and how they fit into the GNN universe
- Understanding convolution and how is it applied to graphs and graph learning
- Implementing convolutional GNNs in a node-prediction problem

In Part 1 of the book, we explored fundamental concepts related to graphs and graph representation learning. All of this served to set us up for Part 2, where we will explore distinct types of GNN architectures, including convolutional GNNs, Graph Attention Networks, and Graph Auto-Encoders.

In this chapter, our goal is to understand and apply Graph Convolutional Networks (GCN) [Kipf] and GraphSage [Hamilton]. These two architectures are part of a larger class of GNNs that are based on applying convolutions to graph data when doing deep learning. Convolutional operations are relatively common in deep learning models, particularly for image problems where the convolutional neural network (CNN) architecture has shown great performance. These operations can be understood as performing a spatial or local averaging. For example, in images, convolution neural networks form representations at incrementally larger pixel subdomains.

In this chapter, we'll be learning more about message passing which we also covered in Chapter 3. Message passing can be understood as a form of convolution but applied to the neighbourhood of the nodes instead of a neighbourhood of pixels. Through convolution operators, we're able to take in more information about our data and this makes our representations better and our downstream applications more accurate. Here, we'll be applying convolutional GNNs to a problem of node prediction. As we discussed in Chapters 1 and 2, node prediction involves inferring certain properties about

the nodes in our graph. It makes sense that convolution is important here, because we want to learn information about the node using information contained in the nodes neighbourhood. For example, ifs we want to learn the age of someone in a social network, we consider the ages of their friends.

In this chapter, we want to first acquaint you with the concepts underlying convolutional GNNs, including convolution itself, convolutions applied to learning on graphs, and some theoretical background of GCN and GraphSage. These two architectures are important to understand because they have been applied to a wide variety of node- and graph-learning problems, and are used widely as baselines for such problems.

Note

While the name *GraphSage* refers to a specific individual architecture, it may be confusing that the name *GCN* also refers to a specific architecture and not the entire class of GNNs based on convolutions. So, in this chapter, I will use *convolutional GNNs* to refer to this entire class of GNNs, which include GraphsSage and GCN. I will use *GCN* or *graph convolutional networks* to refer to the individual architecture introduced in the Kipf paper.)

After you understand the general underlying principles, we will put them to work solving an example task of predicting the categories of Amazon.com products using GCN and GraphSage. This is essentially a node prediction problem, and we will use the Open Graph Benchmark dataset ogbn-products to solve it.

This will be done in the following sections and using code in our Github repo (https://github.com/keitabroadwater/gnns_in_action). Section 4.1 will cover background and theory of convolutional GNNs. Section 4.2 will introduce the example problem. Section 4.3 will explain how a GNN solution is implemented in code. Code snippets will be used to explain the process, but the majority of code and annotation will be in the repo. Finally in Section 4.4, we briefly discuss the use of benchmarks in solving such problems.

4.1 Five Important Concepts for Convolutional GNNs

The GNNs we will use: GCN and GraphSage

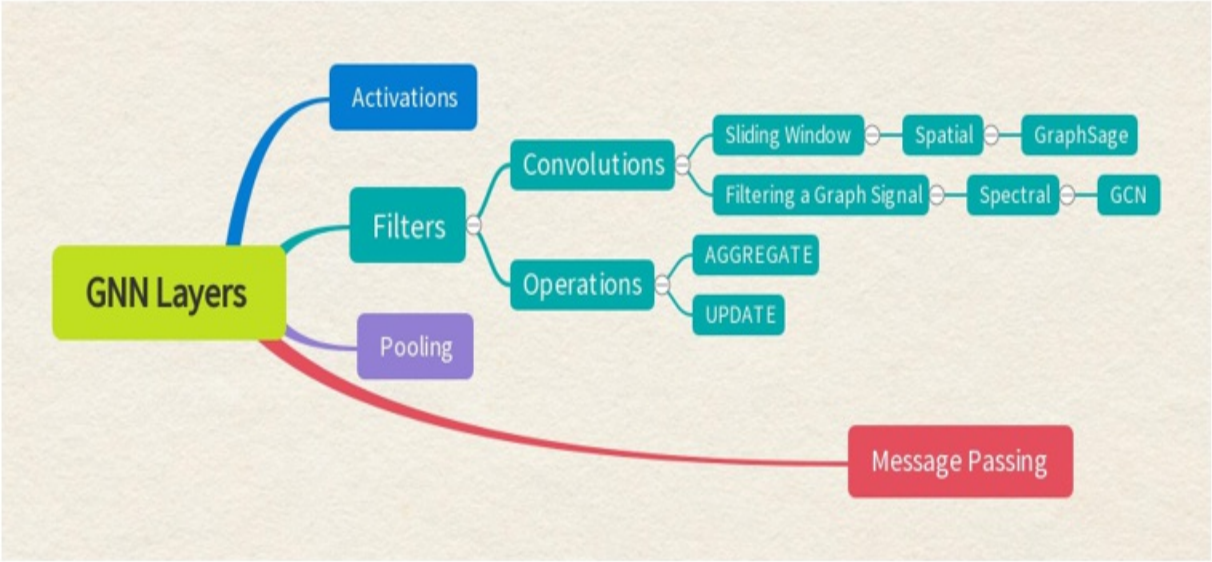
Where do GraphSage and GCN fit in the universe of GNNs? And why are they in the same chapter? Broadly speaking, this is because both GraphSage and GCN apply convolutions in their message passing update step. Before we get into the details of this, we want to briefly provide some context and background by touching on a few technical concepts related to convolutions and GNNs.

If you can understand these two GNNs, you will be able to pick up on many GNN variants that either build upon or use similar elements. Hence, it's very important that we build a solid foundational understanding of these concepts. These concepts include:

- GNN layers
- Spectral and spatial convolutional filters
- Convolution, as a type of GNN filter
- Message Passing, a way to implement convolution which provides another perspective of GNNs
- GNNs and GraphSage explained in context of the preceding convolutions

Figure 4.1 shows how some of these concepts relate to one another and how they are introduced in this section.

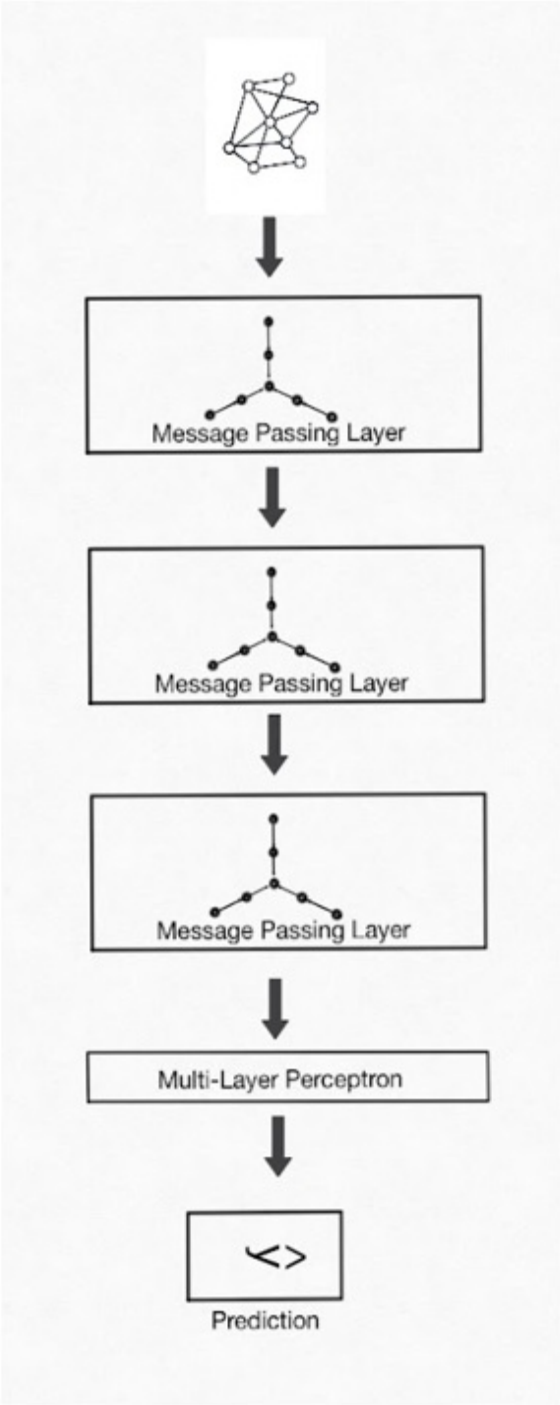
Figure 4.1 Mapping of the concepts described in the section.



Concept 1: Elements of a GNN Layer

Let's dig deeper into the elements of a GNN, and NN in general. In Chapter 3, we introduced the idea of using GNN layers to produce a prediction or create embeddings. Here's that architecture diagram again, reproduced in Figure 4.2.

Figure 4.2 Node embedding architecture diagram from Chapter 3.

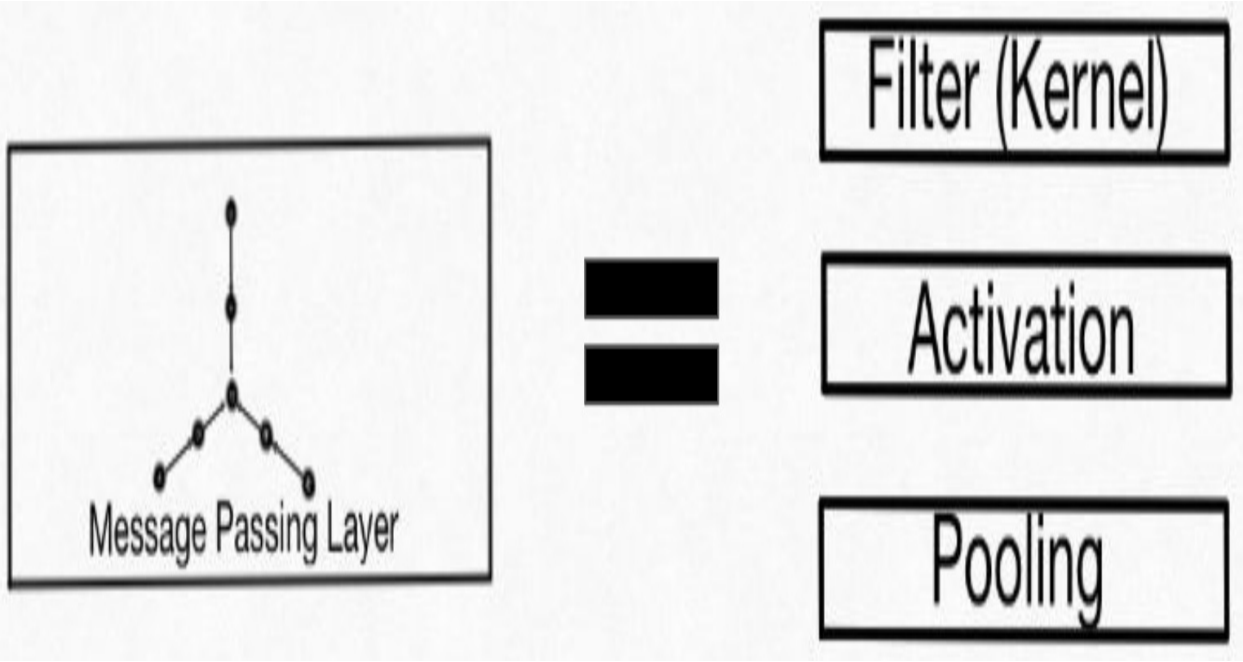


Let’s get below the surface of a GNN layer and examine its elements. Then we’ll tie this to the concept of convolution.

A layer can be interpreted as a sequence of operations that are applied to input data. These operations are

Layer: Filter -> Activation Function -> Pooling

Figure 4.3 Elements of our message passing layer. Each message passing layer consists of a filter, an activation, and a pooling layer.



The output of each entire layer would be a set of node embeddings. These operations consist of:

- Filter (or Kernel) - A process that transforms the input data. Here, our filter will be used to highlight some specific features of the input data and will consist of learnable weights that are optimized by our objective or loss function.
- Activation Function - A non-linear transformation applied to the filter output
- Pooling - An operation that reduces the size of the filter output for graph-level learning tasks. For node-level learning, this part can be omitted.

As we explore different GNNs in this book, we'll return to this set of operations, as most types of GNNs can be seen as modifications of these elements.

The next section introduces a special type of filter, the convolutional filter.

Concept 2: Convolution Methods for Graphs

GCN and GraphSage share a common foundation: the concept of convolution. At a high level and from the context of neural networks, convolution is all about learning by establishing *hierarchies of localized patterns* in the data. Whether we are talking about convolutional neural networks (CNNs or convnets) for image classification or graph convolutional networks (GCNs), convolution-driven processes use layers (the hierarchy part) of such filters that concentrate on a set of nearby image pixels or graph nodes (the *localized* part).

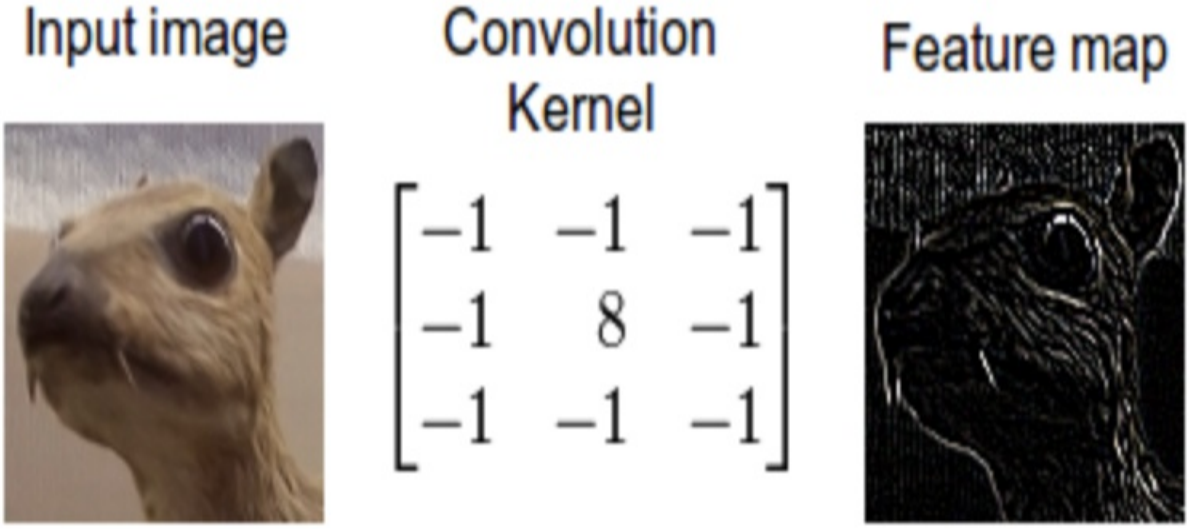
I refer to the *concept* of convolution above because the convolution can be implemented in different ways. Theoretically, convolution relates to the mathematical convolution operator, which we'll be discussing in more detail later in this chapter.

For GNNs, convolution is tackled with many different methods. Let's looks at two of these methods:

- Sliding a window (filter) across a graph
- Filtering a graph signal

"Sliding Window" Methods. In traditional deep learning, convolutional processes learn data representations by applying a special filter called a convolutional kernel to input data. This kernel is smaller in size than the input data, and is applied by moving it across the input data. This is shown in Figure 4.4, where we apply our convolutional kernel (the matrix in the centre) to an image of a rodent. The resulting image has been inverted due to our convolutional kernel. We can see that some of the features have been emphasises, such as the outline of the rodent. This highlights the 'filtering' aspect of convolutions.

Figure 4.4 A convolution of an input image (left). The kernel (middle) is passed over the image of an animal, resulting in a distinct representation (right) of the input image. In a deep learning process the parameters of the filter (the numbers in the matrix) are learned parameters.



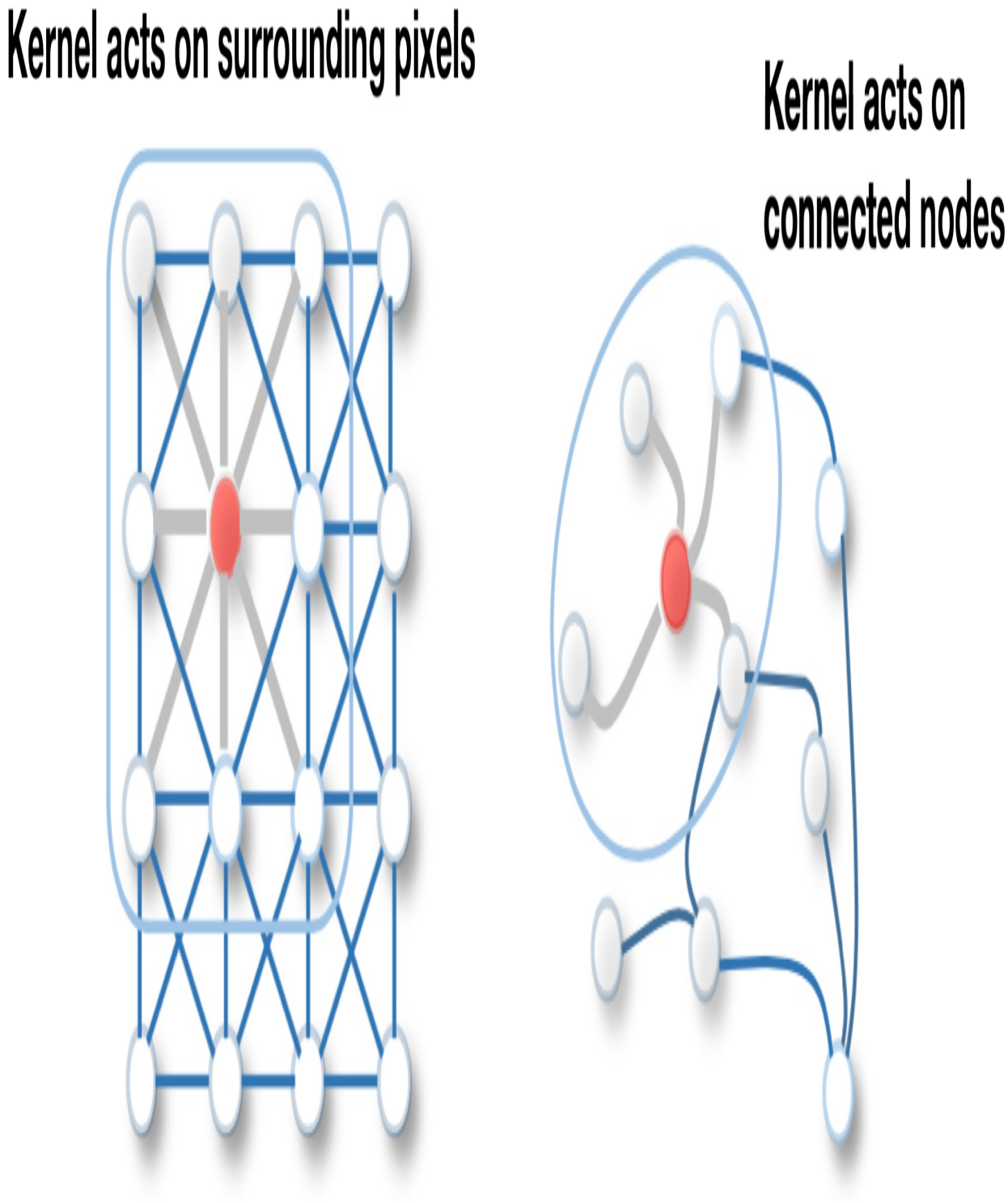
This use of convolutional networks is particularly common in the computer vision domain. For example, when learning on 2-D images we can apply a simple convolutional neural network of a few layers. In each layer, we pass a 2-D filter (kernel) over each 2-D image. The 3x3 filter works on an image many times its size. We can produce learned representations of the input image by doing this over successive layers.

For graphs, we want to apply this same idea of moving a window across our data, but now we need to make adjustments to account for the relational and non-Euclidian topology of our data. With images, we are dealing with rigid 2D grids. With graphs we are dealing with more amorphous data with no rigid shape or order. Without a predefined ordering of the nodes in a graph, we use the concept of a **neighborhood,** consisting of a central node, and all its 1-hop neighbors (that is all nodes within one hop from the central node). Then our sliding window moves across a graph by moving across its node neighborhoods.

In Figure 4.5, we see an illustration comparing convolution applied to grid data and graph data. While in the grid case pixel values are filtered around the nine pixels immediately surrounding the central pixel (marked with a red dot). However, for a graph, node attributes are filtered based on all nodes that can be connected by one edge. Once we have the nodes that we'll be considering, we then need to perform some operation on the nodes. This is

known as the *aggregation operation*; for example, all the node weights in a neighborhood might be averaged, summed or we might take the max value. What is important for graphs is that this operation is *permutation equivariant*. The order of the nodes shouldn't matter.

Figure 4.5 A comparison of convolution over grid data (such as a 2-D image), and over a graph (credit).

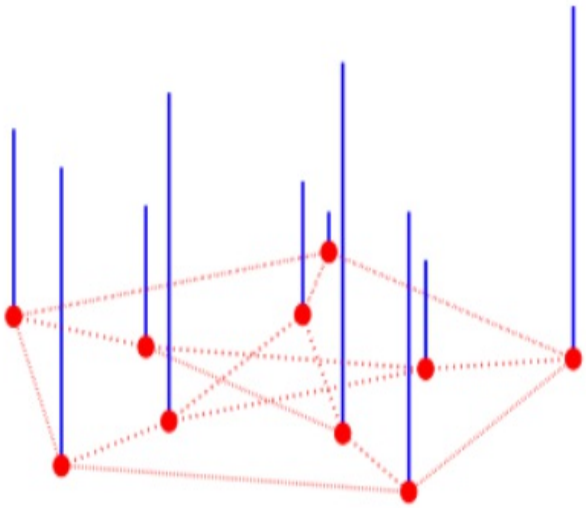


Methods that filter a Graph Signal. To introduce the second method of convolution, let’s examine the concept of a graph signal. In the field of

information processing, signals are sequences that can be examined in either time or frequency domains. When studying a signal in the time domain, we consider its dynamics, namely how it changes over time. From the frequency domain, we consider how much of the signal lies within each frequency band.

We can also study the signal of a graph in an analogous way. To do this, we define a **graph signal** as a vector of node features. Thus, for a given graph, its set of node weights can be used to construct its signal. As a visual example, in Figure 4.6 we have a graph with values associated with each node, where the height of each respective bar represents the combination of node features [Shuman].

Figure 4.6 A random positive graph signal on the vertices of the graph. The height of each blue bar represents the signal value at the node where the bar originates. [Shuman]



To operate on this graph signal, we represent the graph signal as a 2D matrix, where each row is a set of features associated with a particular node.

Using a graph signal, we can apply operations from signal processing on the graphs. One critical operation is that of the Fourier transform. The Fourier transform can express a graph signal, its set of node features, into a frequency representation. Conversely, an inverse Fourier transform will revert the frequency representation into a graph signal.

So, with these notions of a graph signal, its matrix representation, and Fourier

transforms, we can summarize this second method of convolutions on graphs. In a mathematical form, the convolutional operation is expressed as an operation on two functions that produces a third function:

(4.1)

$$\text{Convolution} = g(x) = f(x) \odot h(x)$$

Where f(x) and h(x) are functions and the operator represents element-wise multiplication. In the context of CNNs, the image and the kernel matrices are the functions in Equation 4.1:

(4.2)

$$\text{Convolution} = \text{image}() \odot \text{filter}()$$

This mathematical operation is interpreted as the kernel sliding over the image, as in the sliding window method.

To apply the convolution of Equation 4.1 to graphs, we use the following ingredients:

- Use matrix representations of the graph
  - Vector **x** as the graph signal
  - Adjacency matrix **A**
  - Laplacian matrix **L**
  - A matrix of eigenvectors of the Laplacian **U**
- Use a parameterized matrix for the weights, **H**
- Apply Fourier Transform Using matrix Operations $\mathbf{U}^T\mathbf{x}$

This leads to the expression for convolution over a graph:

(4.3)

$$\text{Graph Convolution} = \mathbf{x} *_G \mathbf{H} = \mathbf{U} (\mathbf{U}^T\mathbf{x} \circ \mathbf{U}^T\mathbf{H})$$

Since this operation is not a simple element-wise multiplication, we are using the symbol $*_G$ to express this operation. Several convolutional-based GNNs

build on Equation 4.3; below we will examine the GCN version.

Above and Beyond: Limitations of Traditional Deep Learning Methods to Graphs

Why can't Equation 4.1 be applied to a graph structure, that is why can't we simply apply the same convnet described above to a graph? The reason is because graph representations have an ambiguity that image representations don't. Convnets, and traditional deep learning tools in general, are not able to resolve this ambiguity. A neural network that can deal with this ambiguity is said to be **permutation equivariant** or **permutation invariant**.

Let's illustrate the ambiguity of a graph vs an image by considering the image of the rodent above. A simple representation of this set of pixels is as a 2D matrix (with dimensions for height, width). This representation would be unique: if we swap out two rows of the image, or two columns, we don't have an equivalent image. Similarly, if we swap out two columns or rows of the matrix representation of the image (as shown in Figure 4.7), we don't have an equivalent matrix.

Figure 4.7 The image of a rodent is unique. If we swap out two columns (right image), we end up with a distinct photo with respect to the original.

This is not the case of a graph. Graphs can be represented by adjacency matrices (which we describe in Chapter1 and Appendix A), such that each row and column element stands for the relation between two nodes. If an element is non-zero, it means that the row-node and column node are linked.

Given such a matrix, we can repeat our experiment above and swap out two rows as we did the image. Unlike the case of the image, we end up with a matrix that represents the graph we started with. We can do any number of permutations or rows and columns and end up with a matrix that represents the same graph.

Returning to the convolution operation, to successfully apply a convolutional filter or a convnet to the graph's matrix representation, such an operation or layer would have to yield the same result no matter the ordering of the adjacency matrix (since every ordering describes the same thing). Convnets fail in this respect.

Finding convolutional filters that can be applied to graphs has been solved in a variety of ways. In this chapter we will examine two ways this has been done: spatial and spectral methods. For a deeper discussion and derivation of convolutional filters applied to graphs, see *Hamilton*.

Concept 3: Spectral vs Spatial Methods

In the last section, we talked about interpreting convolution via a thought experiment of sliding a window filter across part of a graph consisting of a local neighborhood of linked nodes. We also interpreted convolution as processing graph signal data through a filter.

It turns out that these two interpretations highlight two branches of convolutional graph neural networks: spatial and spectral methods. 'Sliding window' and other methods that rely on a graph's geometrical structure to perform convolution are known as **spatial methods**. Graph signal filters are grouped as **spectral methods**.

It should be said that there is no clear demarcation between spectral and spatial methods, and often one type can be interpreted as the other. For example, one contribution of GCN is that it demonstrated that its spectral derivation could be interpreted in a spatial way.

Practical Differences Between Spatial and Spectral Methods. At the time of writing, spatial methods are preferred since they have less restrictions and, in

general, offer less computational complexity. We've highlighted these in Table 4.1.

Table 4.1 A comparison of spectral and spatial convolutional methods.

Spectral	Spatial
Operation: performing a convolution using a graph's eigenvalues.	Operation: aggregation of node features in node neighborhoods
· Graphs must be undirected · Operation relies upon node features · Generally less computationally efficient	· Undirectedness not a requirement · Operation not dependent upon node features · Generally more computationally efficient

Concept 4: Message Passing

All graph neural networks operate under the process of updating and sharing node information across the network. This process and computation is known as message passing. For convolutional networks, in addition to the convolutional frameworks described above, we can view these GNNs from the perspective of the message passing operation.

Let's re-examine the GNN layer from above, adding more detail. Before, we said that a GNN layer contained a filter, an activation function, and a pooling operation, and that this layer serves to update an embedding. Let's drill down on the filter element. We can break down the filter into two operations, which we will call *AGGREGATE-NODES* and *UPDATE-EMBEDDING*.

Thus, a second way to interpret a GNN layer is:

Layer: Filter(UPDATE-EMBEDDING + AGGREGATE-NODES) -> Activation Function -> Pooling

Let's discuss these in more detail. The filter now consists of two additional operations. First we *aggregate nodes* using our aggregation operator. For example, we might sum the features, average them, or choose the maximum values. The most important thing is that the order of the nodes shouldn't matter for the final representation. Once we have aggregated information from all nodes, or collected all the messages, we then *update our embedding.* This is done by passing the new messages through a neural network and combining this with the previous embeddings.

The activation function is a non-linear transformation that is applied to the filter output. These are standard activation functions used in artificial neural networks, such as the Rectified Linear Unit (or ReLU) which is the maximum value between zero and the input value. The pooling step then reduces the overall size of the filter output for any graph-level learning tasks. For node-prediction, this can be omitted, which we'll do here.

Hence, we can follow Hamilton(ref), and express the message passing operation for as:

(4.4)

Updated Node Embeddings = $h^{(k)}_u$ = UPDATE = σ (Wa * h_u + Wb * AGGREGATE)

where the aggregation operator here might just be summing all the different embeddings,

(4.5)

AGGREGATE = $\sum_{V \in N(u)} h_v$

and where $h_u^{(k)}$ is the updated embedding for the kth layer and for node *u*, h_v is an embedding for a node in node u's neighborhood. Wa and Wb are learnable weights. σ is an activation function.

With the concept of message passing, we also adjust the concept of a GNN layer. For the message passing formulas above, we see that a node and its neighborhood play a central part. Indeed, this is one of the main reasons that GNNs have proven to be so successful.

If we run the AGGREGATE and UPDATE operations only once, or $k=1$ times, we are aggregating the information from the neighbors one hop away from our central node. If we run these operations for two iterations, or k=2, we aggregate nodes within two hops of our central node. Thus, the number of GNN layers is directly tied to the size of the neighborhoods we are interrogating with our model.

Concept 5: GCNs and GraphSage

Given the above, we can now dive into how GCN and GraphSage models work.

GCN is a spectral-based GNN that builds upon several improvements to the convolution equation (4.3) to simplify operations and to reduce computational cost. These involve using a filter based on a polynomial rather than set of matrices, and to limit the number of hops to k=1. In terms of computational complexity it reduces the computational complexity from a quadratic to a linear complexity, which is significant.

Looking at the GCN from the message passing point of view, we also see adjustments from the equations above. One adjustment is to replace the UPDATE operation (Equation 4.4) with an AGGREGATE operation that involves self loops (recall from Appendix A that a self loop consists of a node with an edge that connects back to the node itself). We see that Equation 4.4 has two terms, the first of which involves updating the embeddings of the central node, u. For a GCN, we eliminate that first term, and adjust only the second term (the AGGREGATE term) so that it involves not only the neighborhood of a node, but also the node itself.

Another adjustment involves **normalizing** the AGGREGATE operation. In Equation 4.5, aggregation of the embeddings is a sum. This can lead to problems in graphs where there are nodes where the degree of nodes has high

variance. If a graph contains nodes whose degrees are high, those nodes will dominate a summed AGGREGATION. In using normalization to solve this, one method is to replace summing (Eq. 4.5) with averaging (Eq. 4.6, below).It's AGGREGATION function is expressed:

(4.6)

$$m_{N(u)} = \sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}$$

With these adjustments, we have the GCN layer:

(4.7)

$$\text{GCN Updated Node Embeddings} = h_u^{(k)} = \sigma(W^{(k)} \sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}$$

GCN has proven to be an effective and popular GNN layer to build upon. However, since the aggregation in Equation 4.7 happens over the entire neighborhood of each node, it can be computationally expensive to use, especially for graphs with nodes that have degrees over 1000. **GraphSAGE** improved upon this by limiting the amount of neighboring nodes that are aggregated during an iteration. It aggregates from a randomly selected sample from the neighborhood. The aggregation operator is flexible (e.g. it can be a sum or average) but the messages that are considered are now only a subsample of all messages. This layer can be expressed as:

(4.8)

$h(\mathrm{k}) = \sigma(W(k) \cdot \boldsymbol{f}\,(h(\mathrm{k}-1), \{h(\mathrm{k}-1), \forall u \in S\}))$

Where $\boldsymbol{f}$ is the aggregation function and $\forall u \in S$ denotes that the neighborhood is picked from a random sample, S of the total sample of messages/neighbours.

4.2 Problem Description: Predicting Consumer Product Categories

Having reviewed convolution and how this relates to GNNs and message passing, let’s next examine how to put these concepts into practice. Our problem for this chapter explores the domain of online retail and product relationships. We’re going to be using data about customer purchasing to predict product categories.

Whether shopping in a local department store or browsing online stores, it’s helpful when a store understands how individual products or categories of products are related to another. One use of this information is to make the shopping experience easier by having similar items placed closer to one another in the store or online. A good store manager wouldn’t place shoe accessories too far away from shoes, or printer supplies too far away from printers.

Figure 4.8 Illustration of digital shopping with product categories.


24
18
@
14
33
12
22
8

Online stores are able to point to similar items by using a link hierarchy (e.g., having a clothing page as a parent page, a shoe specific page as a child, and a shoe accessories page a child to the shoe page). They also help consumers find related items by devoting sections of a webpage to 'related items' and '*others users also looked at*'.

For certain product relationships, one could argue that machine learning is overkill for this task, though for stores with millions of products, the scale could necessitate some automated solution. But what about less obvious product relationships? For example, we might want to pair basketball shoes with a basketball or joint and muscle pain relief near to running shoes. Less obvious still, we might even want to pair basketball shoes to a newly released basketball movie.

Sometimes, non-obvious product relationships are established via customer behavior; innovative consumers discover an appealing pairing which appeals to the crowd. If such innovations are popular enough, they may lead to new product categories. An example of a surprising product pairing is coffee and dry shampoo used in mid-day physical workouts: coffee is used to enhance the workout, while dry shampoo is used to quickly groom afterward.

Problem Definition: Identify a Product's Category

Our Dataset: The Amazon Product Co-Purchasing Network

To explore the product relationships, we will use the Amazon product co-purchasing graph, a dataset of products that have been bought together in the same transaction (which we will define as a co-purchase). In this dataset, products are represented by nodes, while co-purchases are represented by vertices. The dataset, *ogbn-products*, consists of 2.5 million nodes, and 61.9 million edges (a larger version of the dataset, *ogbn-products100M*, contains 111 million nodes and 1,616 million edges). This means that this dataset consists of 2.5 million products, and 61.9 established co-purchases.

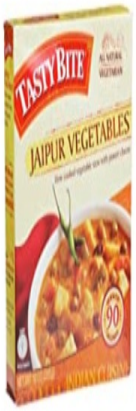
The construction of this dataset is a long journey in itself, very much of interest to graph construction, and the decisions that have to be made to get a

meaningful and useful dataset. Put simply, this dataset was derived from purchasing log data from Amazon, which directly showed co-purchases, and from text data from product reviews, which was used to indirectly show product relationships. For the in-depth story, I refer you to [McCauly].

Figure 4.9 Examples of co-purchases on Amazon.com. Each product is represented by a picture, a plain text product title, and a bold text product category. We see that some co-purchases feature products that are obvious complements of one another, while other groupings are less so.

Radar Detector
Electronics

Radar Detector
Car Mount
Electronics



Jaipur Vegetables
Gourmet Food

Bengal Lentils
Gourmet Food

Punjab Eggplant
Gourmet Food

Portable Music Player
Electronics

Europe Travel Books
Books

Green Logger Shirt
Clothes

Men's Jeans
Clothes

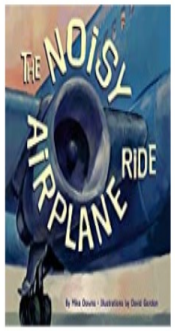
Black Pleated
Khakis
Clothes

Navy Blue
Henley
Clothes

Adzuki Beans
Gourmet Food

Wireless Speaker
Electronics

Bluetooth Dongle
Electronics



Children's Books
Books

To further illustrate the concept of co-purchases, in Figure 4.9 we show six example co-purchase images for an online customer. For each product, we include a picture, a plain text product label, and a bold text category label.

Some of these co-purchase groups seem to fit together well, such as the book purchases, or the clothing purchase. Other co-purchases are less explainable, such as an Apple Ipod being purchased with instant meals, or beans being purchased with a wireless speaker.

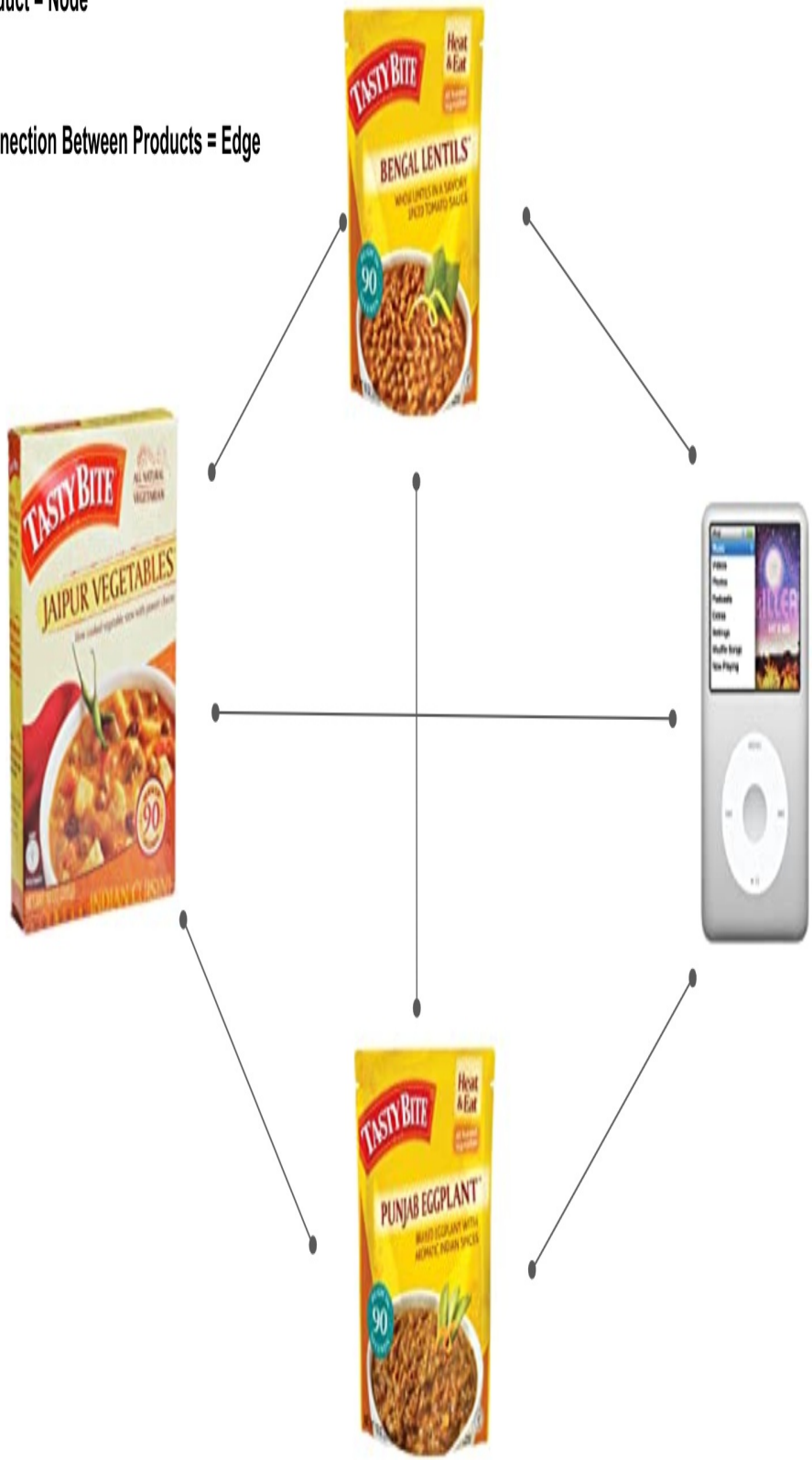
In those less obvious groupings, maybe there is some latent product relationship. Or maybe it's just mere coincidence. Examining the data at scale can provide clues.

To show how the co-purchasing graph would appear at a small scale, Figure 4.10 takes one of the images above and represents the products as nodes, with the edges between them representing each co-purchase. For one customer and one purchase, this is a small graph, with only four nodes, and six edges. But for the same customer over time, or for a larger set of customers with the same tastes in food, or even all of the customers, it's easy to imagine how this graph can scale with more products and product connections branching from these few products.

Figure 4.10 A graph representation of one of the co-purchases from figure 4.x. Each product's picture is a node, and the co-purchases are the edges (shown as lines) between the products. For the 4 products shown here, this graph is only the co-purchasing graph of one customer. If we show the corresponding graph for all customers of Amazon, the number of products and edges could feature tens of thousands of product nodes and millions of co-purchasing edges.

Product = Node

Connection Between Products = Edge



GNN interpretation of problem: A semi-supervised classification problem

We will use GCN and Graphsage to predict the categories of these products. For this purpose, there are 47 categories that will be used as targets in a classification task.

Why is such a prediction task important? Being able to categorize a particular product with a model can aid a general categorization task at scale, and be used to find non-intuitive categories and product relationships. We can imagine these categories representing aisles in the physical store or possible ways to group items for online offers.

We will cover some initial data preprocessing below but note that much of the heavy lifting has already been done. The dataset (with included dataloaders) has been provided by Open Graph Benchmark product, with an usage license from Amazon.

4.3 Implementation in PyG

In this section, we will solve the node-classification problem outlined in the last section using the architectures explained in Section 4.1: Kipf's GCN and GraphSage. The details and full code are available in our github repository. Here, we will highlight the major components of a solution.

We will also give more detail to how GCN and GraphSAGE are implemented in PyTorch geometric. We'll follow the following steps in this chapter:

- Preprocess - Prep the data
- Define model - Define the architecture
- Define training - Set of the training loop and learning criteria
- Train - Execute the training process
- Model Saving - Save the model
- Inference - Use the saved model to make predictions

Needed Software

- Pytorch and Pytorch Geometric (PyG) - Deep learning frameworks where many of the tasks needed to develop deep learning models for graphs have been done. Many popular GNN layers (such as GCN and GraphSage) have been implemented in PyG. (Documentation: pytorch: https://pytorch.org/docs/stable/index.html ; PyG: https://pytorch-geometric.readthedocs.io/en/latest/index.html)
- Matplotlib - A standard visualization library for python. We will use it in our data exploration.

- Pandas and Numpy - Standard data munging and numerical packages in python. Pandas is useful in data exploration, while numpy is used often in calculations and data transformations in our data pipelines.
- NetworkX - A basic graph processing system, which we will use for data exploration. (Documentation: https://networkx.org/documentation/stable/index.html)
- OGB - Library of the Open Graph Benchmark project. Allows us to access and evaluate datasets, including the Amazon products dataset, used in our example. (Documentation: https://ogb.stanford.edu/docs/home/)

Most of these can be installed with the simple pip install command. Examples showing the installation of these packages can be found in our repository (link in references).

4.3.1 Pre-processing

The Amazon product co-purchasing dataset can be accessed via the Open Graph Benchmark (OGB). The *PygNodePropPredDataset* function from the ogb library downloads this dataset into a folder of your choosing. The zip file containing the dataset and associated files is 1.38GB.

```
dataset = PygNodePropPredDataset('ogbn-products')
```

Once downloaded and decompressed, we find that the file contains the following directories:

- mapping - files that contain metadata about the dataset: human-understandable labels/categories for the label indices, and Amazon product numbers for the node indices
- processed - files that allow downloaded data to be preprocessed into a PyG format
- raw - various formats of the graph data, including edge lists, and node features and labels.
- split - convention used to split the data into train/validation/test sets

To get the training/validation/test splits for the data, we can use the *get_idx_split*() method.

```
split_idx = dataset.get_idx_split()
```

Another feature of OGB datasets is that they come with an evaluator. An evaluator is a curated performance metric tailored for that specific dataset. This is provided by the OGB database and allows for easy comparison of different GNN and graph-based inference methods on the data. For our dataset, this evaluation metric will be accuracy. The evaluator can be used when training models, and comparing models. It can be called with the *Evaluator* function.

```
evaluator = Evaluator(name='ogbn-products')
```

Light Data Exploration

Once downloaded, we can perform some exploratory data analysis (EDA) to get a feel of the data. In Figure 4.8, we examine the distribution of categories in the dataset.

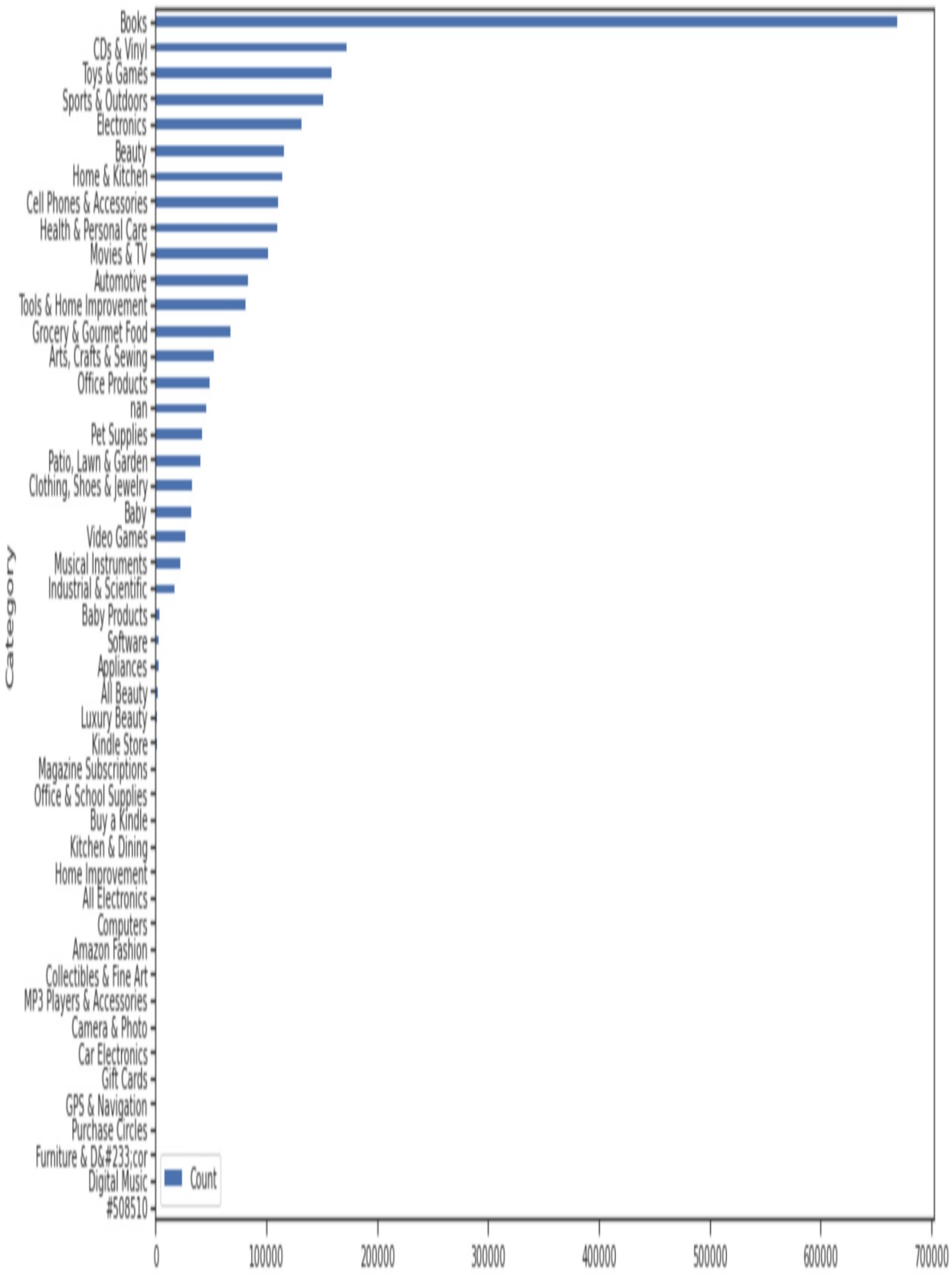
To see the distribution of categories in the graph, we pull the label data from the graph, and we use the category metadata in the downloaded *dataset* folder.

Listing 4.1 Light Data Exploration

```
# Putting the category metadata into a dataframe

path_to_file = '/content/dataset/ogbn_products/mapping/labelidx2p
df = pd.read_csv(path_to_file)    #A

y = data.y.tolist()    #B
y = list(flatten(y))     #B
count_y = collections.Counter(y)    #B

index_product_dict = dict(zip(df['label idx'], df['product catego
products_hist = dict((index_product_dict[key], value) \     #C
                for (key, value) in dict(count_y).items())   #C

category_df = pd.DataFrame(products_hist.items(), columns=['Categ
category_df = category_df.set_index('Category')    #D
category_df = category_df.sort_values('Count')    #D
category_df.plot(kind='barh')    #D
```

In Figure 4.11, we see that the categories with the highest counts of nodes are *Books* (668950 nodes), *CDs & Vinyl* (172199 nodes), and *Toys & Games* (158771 nodes). The lowest are *Furniture and Decor* (9 nodes), *Digital Music* (6 nodes), and an unknown category with one node (Category *#508510*). We also observe that many categories have very low proportions in the dataset. The mean count of nodes per label/category is 52,107; the median count is 3,653.

Figure 4.11 Distribution of node labels generated using Listing 4.1.


Category
Books
CDs & Vinyl
Toys & Games
Sports & Outdoors
Electronics
Beauty
Home & Kitchen
Cell Phones & Accessories
Health & Personal Care
Movies & TV
Automotive
Tools & Home Improvement
Grocery & Gourmet Food
Arts, Crafts & Sewing
Office Products
nan
Pet Supplies
Patio, Lawn & Garden
Clothing, Shoes & Jewelry
Baby
Video Games
Musical Instruments
Industrial & Scientific
Baby Products
Software
Appliances
All Beauty
Luxury Beauty
Kindle Store
Magazine Subscriptions
Office & School Supplies
Buy a Kindle
Kitchen & Dining
Home Improvement
All Electronics
Computers
Amazon Fashion
Collectibles & Fine Art
MP3 Players & Accessories
Camera & Photo
Car Electronics
Gift Cards
GPS & Navigation
Purchase Circles
Furniture & Décor
Digital Music
#508510
Count
0
100000
200000
300000
400000
500000
600000
700000

4.3.2 Defining the model

In this section, we want to first understand how the message passing method, described in Section 4.2, is implemented in Pytorch Geometric. With an understanding of the GCN and GraphSAGE layers, we will explain how the GNN architecture is coded. This explanation builds on the previous sections, as well as how we created embeddings in Chapter 3.

Implementation of GCN Message Passing in Pytorch Geometric

Recall in section 4.2, we defined message passing for GCN as:

(4.9)

$$\text{GCN Updated Node Embeddings} = h_u^{(k)} = \sigma(W^{(k)} \sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}})$$

where

h is the updated node embedding

sigma, , is a non-linearity (i.e., activation function) applied to every element

W, is a trained weight matrix

|*N*| denotes the count of the elements in the set of graph nodes

The summed factor

$$\sum_{v \in N(u)} \frac{h_u}{\sqrt{|N(u)||N(v)|}}$$

is a special normalization called symmetric normalization.

Also, recall that GCN does not use an UPDATE operation; it instead uses an

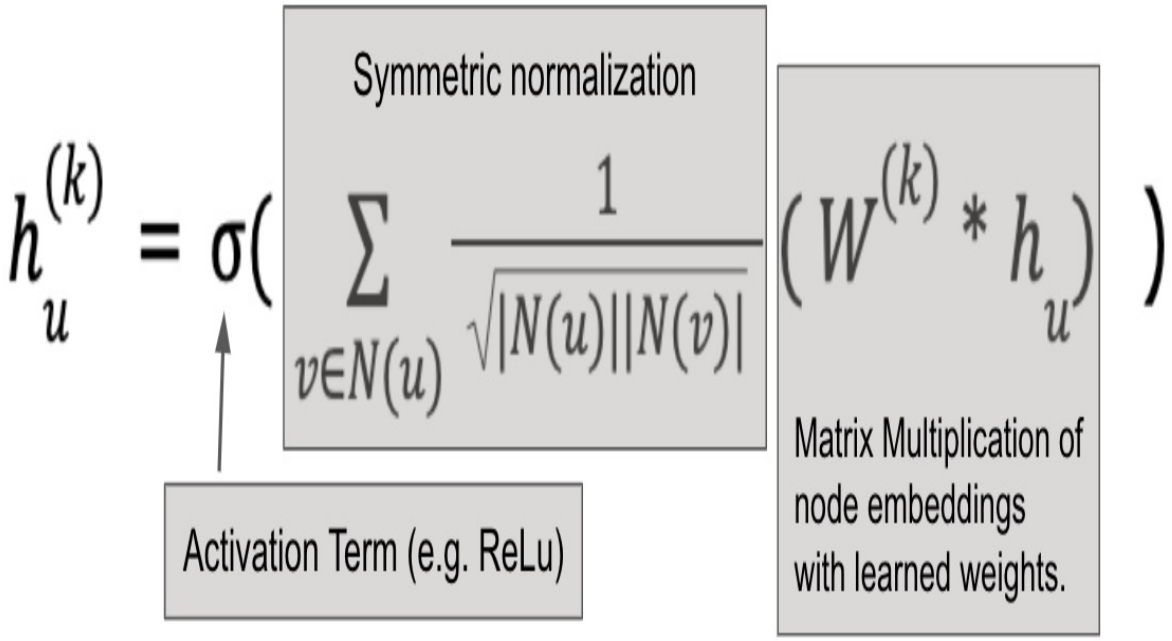
AGGREGATION function on a graph with self-loops.

So, to implement GCN, the following operations must occur:

- Graph nodes must be adjusted to contain self loops
- Matrix multiplication of the trained weight matrix and the node embeddings
- Normalization operations: Summing the terms of the symmetric normalization

In Figure 4.12, we explain each of the terms used in a message passing step in detail.

Figure 4.12 Mapping of key computational operations in the GCN embedding formula.



In the PyTorch Geometric (PyG) documentation, one can find source code that implements the GCN layer, as well as a simplified implementation of the GCN layer. Below, we'll point out how the source code implements the

above key operations.

Table 4.2 describes how functions that the PyG source code implements relates to terms in Figure 4.12.

Table 4.2 Mapping of key computational operations in the GCN embedding formula.

Operation	Function/Method
Add self loops to nodes	`gcn_norm()`, annotated, below
Multiply weights and embeddings $W^{(k)}*h_u$	`GCNConv.__init__; GCNConv.forward`
Symmetric Normalization	`gcn_norm()`, annotated, below

These operations are implemented using a class and a function:

- A function, *gcn_norm*, which performs normalization and add self loops to the graph
- A class, *GCNConv*, which instantiates the GNN layer and performs matrix operations

In Listings 4.2 and 4.3, we show the code in detail for the function and class and use annotation to highlight the key operations.

Listing 4.2 The GCN Norm Function

```
def gcn_norm(edge_index, edge_weight=None, num_nodes=None, improv
             add_self_loops=True, dtype=None): #A

   fill_value = 2. if improved else 1. #B
```

```
    if isinstance(edge_index, SparseTensor): #C
        adj_t = edge_index
        if not adj_t.has_value():
            adj_t = adj_t.fill_value(1., dtype=dtype)
        if add_self_loops:
            adj_t = fill_diag(adj_t, fill_value)
        deg = sparsesum(adj_t, dim=1)
        deg_inv_sqrt = deg.pow_(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0.
        adj_t = mul(adj_t, deg_inv_sqrt.view(-1, 1))
        adj_t = mul(adj_t, deg_inv_sqrt.view(1, -1))
        return adj_t

    else: #C
        num_nodes = maybe_num_nodes(edge_index, num_nodes)

        if edge_weight is None:
            edge_weight = torch.ones((edge_index.size(1), ), dtype
                                     device=edge_index.device)

        if add_self_loops:
            edge_index, tmp_edge_weight = add_remaining_self_loops
                edge_index, edge_weight, fill_value, num_nodes)
            assert tmp_edge_weight is not None
            edge_weight = tmp_edge_weight

        row, col = edge_index[0], edge_index[1]
        deg = scatter_add(edge_weight, col, dim=0, dim_size=num_no
        deg_inv_sqrt = deg.pow_(-0.5)
        deg_inv_sqrt.masked_fill_(deg_inv_sqrt == float('inf'), 0)
        return edge_index, deg_inv_sqrt[row] * edge_weight * deg_i
```

Arguments of *gcn_norm* are::

- *edge_index*: the node representations in a Tensor or Sparse Tensor form
- *edge_weight*: an array of 1-dimensional edge weights is optional
- *num_nodes*: a dimension of the input graph
- *improved*: introduces an alternative method to add self-loops from the Graph U-Nets paper, referenced at the end of the chapter.
- *Add_self_loops*: adding self loops is the default, but is optional

In Listing 4.3, we have excerpts of the *GCNConv* class, which calls on the *gcn_norm* function as well as the matrix operations.

Listing 4.3 The GCN Class

```
class GCNConv(MessagePassing):
…
   def __init__(self, in_channels: int, out_channels: int,
                improved: bool = False, cached: bool = False,
                add_self_loops: bool = True, normalize: bool = Tr
                bias: bool = True, **kwargs):
…
       self.lin = Linear(in_channels, out_channels, bias=False,
                         weight_initializer='glorot')
…
    def forward(self, x: Tensor, edge_index: Adj,
                edge_weight: OptTensor = None) -> Tensor:

if self.normalize: #A
            if isinstance(edge_index, Tensor):
                cache = self._cached_edge_index
                if cache is None:
                    edge_index, edge_weight = gcn_norm(
                        edge_index, edge_weight, x.size(self.node
                        self.improved, self.add_self_loops)
                    if self.cached:
                        self._cached_edge_index = (edge_index, ed
                else:
                    edge_index, edge_weight = cache[0], cache[1]
…
        x = self.lin(x) #B

        out = self.propagate(edge_index, x=x, edge_weight=edge_we

        if self.bias is not None: #D
            out += self.bias

        return out
```

Implementation of GraphSage Message Passing in Pytorch Geometric

Again, recall in section 4.2, we defined message passing for GraphSage as:

(4.10)

GraphSage Updated Node Embeddings = $h(k) = \sigma(W(k) \cdot \boldsymbol{f}(h(k-1), \{h(k-1), \forall u \in S\}))$

where

f is the aggregation function (sum, average, or other), and

$\forall u \in S$ denotes that the neighborhood is picked from a random sample, *S*.

If we choose the mean as the aggregation function, this becomes:

(4.11)

$h(\mathrm{k})\ \mathrm{v} \leftarrow \sigma(\mathrm{W} \cdot \mathrm{MEAN}(\{h(\mathrm{k}-1),\mathrm{v}\} \cup \{h(\mathrm{k}-1),\mathrm{u}\ ,\ \forall \mathrm{u} \in N(\mathrm{v})\}$

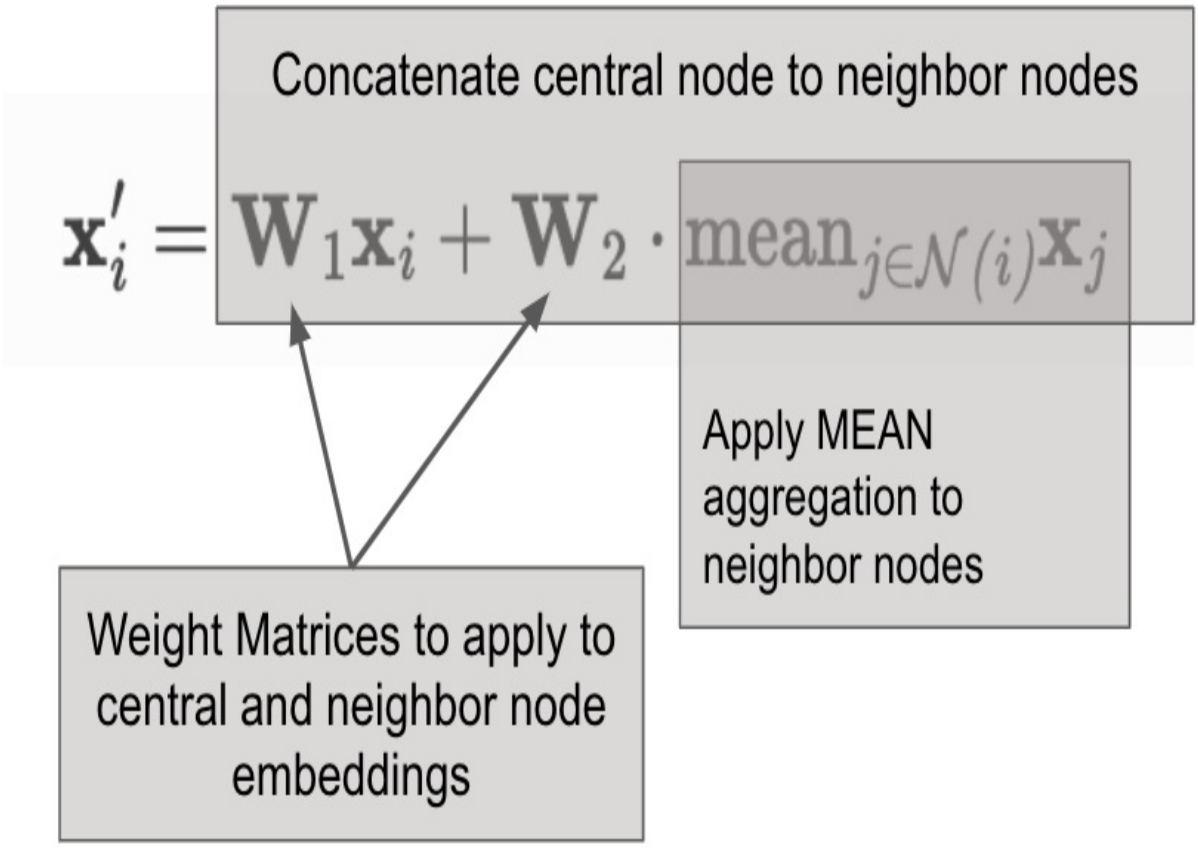
To implement this, we can further reduce this to:

(4.12)

$$\mathbf{x}'_i = \mathbf{W}_i\,\mathbf{x}_i + \mathbf{W}_i \cdot mean_{j \epsilon N(i)}\ \mathbf{x}_j$$

Where $\mathbf{x}_i$**'** denotes the generated central node embeddings, and $\mathbf{x}_i$ and $\mathbf{x}_j$ are the input features of the central and neighboring nodes, respectively. In Figure 4.13, we expand on the meaning of each of the terms in Equation 4.12. Specifically, we set out how central nodes and neighbour nodes have different operations. The weight matrices are applied to both central and neighbours but only the neighbour nodes have an aggregation operator (in this case the mean).

Figure 4.13 Mapping of key computational operations in the GraphSage embedding formula.



With a mean (average) aggregator, these operations are a bit more straightforward compared with those for GCN. From the GraphSage paper, we have the general embedding updating process, which the paper introduces as Algorithm 1, reproduced here in Figure 4.14.

Figure 4.14 Algorithm 1, the GraphSAGE embedding generation algorithm from the GraphSAGE paper.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

1 $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
2 **for** $k = 1...K$ **do**
3 **for** $v \in \mathcal{V}$ **do**
4 $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
5 $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
6 **end**
7 $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
8 **end**
9 $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

The basics of this algorithm can be described as follows:

1. For every layer/iteration (line 2), *and* for every node (line 3):
   a) Aggregate the embeddings of the neighbors (line 4)
   b) Concatenate neighbor embeddings with the central node (line 5)
   c) Matrix multiply that concatenation with the Weights matrix (line 5)
   d) Multiply that result with an activation function (line 5)
   e) Apply a normalization (line 7)
2. Update node features, z, with node embeddings, h

We have now seen all the main features of the GCN algorithm. This includes an aggregation function which is implemented using message passing, a concatenation step, an activation step and finally, the normalization. In Table 4.3, we break down the key operations and where they occur in PyG's

GraphSage class:

Table 4.3 Mapping of key computational operations in the GCN embedding formula.

Operation	Function/Method
Aggregate the embeddings of the neighbors (sum, mean, or other)	SageConv.message_and_aggregate
Concatenate neighbor embeddings with that of the central node	SageConv.forward
Matrix multiply that concatenation with the Weights matrix	SageConv.message_and_aggregate
Apply an activation function	If the *project* parameter set to *True*, done in SageConv.forward
Apply a normalization	SageConv.forward

For GraphSage, PyG also has source code to implement this layer in the *SageConv* class, excerpts of which are shown in Listing 4.4.

Listing 4.4 The GraphSAGE Class

```
class SAGEConv(MessagePassing):
…
   def forward(self, x, edge_index, size):
```

```
         if isinstance(x, Tensor):
             x: OptPairTensor = (x, x)

        if self.project and hasattr(self, 'lin'): #A
            x = (self.lin(x[0]).relu(), x[1])

        out = self.propagate(edge_index, x=x, size=size) #B
        out = self.lin_l(out) #B

        x_r = x[1] #C
        if self.root_weight and x_r is not None: #D
            out += self.lin_r(x_r) #D

        if self.normalize: #E
            out = F.normalize(out, p=2., dim=-1) #E

        return out

   def message(self, x_j):
return x_j

   def message_and_aggregate(self, adj_t, x):
        adj_t = adj_t.set_value(None, layout=None) #F
        return matmul(adj_t, x[0], reduce=self.aggr) #F
…
```

Architecture Construction

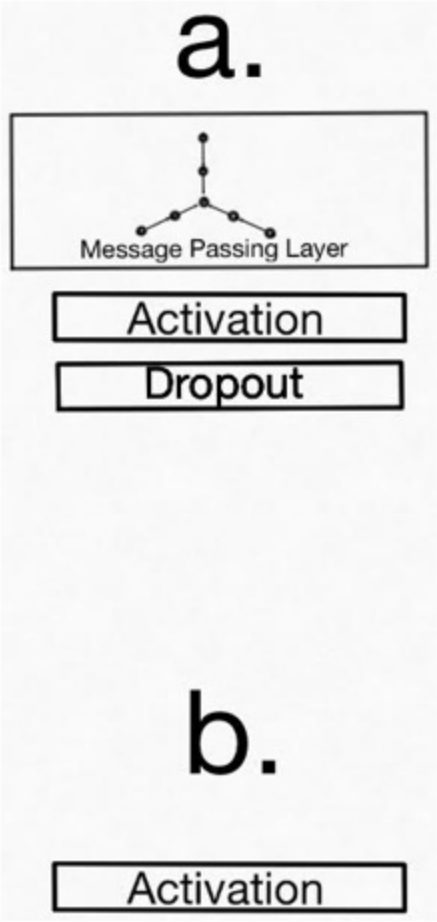
With the GCN and GraphSAGE layers implemented, we can construct architectures which stack these with other neural network layers and functions. This is building on Concept 1 before, namely that we have different layers of a GNN that are combined. For GraphSAGE, we will have a very similar architecture to the one set out in Chapter 3. This consists of:

- Message Passing layer: where information gets passed between edges
- Activation layer: where a non-linear function is applied to the previous output
- Dropout layer: switching off some of the output units.

Note that here, these layers are the same as we describe in Concept 1 of Section 4.1, except we now also include a dropout layer. Dropout layers are useful for preventing overfitting, especially where there is a lot of data. They

work by randomly setting some weights of a neural network layer to zero with some predefined rate for each step during training time.

Figure 4.14 In the architecture of a GCN or GraphSage, each GNN layer (a) consists of a message passing layer, an activation layer, and a dropout layer. An activation layer (b) gets added after each GNN layer.



In PyTorch, neural network architectures are typically created as a class which inherits from the *torch.nn.Module* class. Listing 4.6 shows our class, consisting of:

- An *__init__* method, which defines and initializes parameter weights for our layers.
- A *forward* method, which governs how data passes forward through our architecture.

For this architecture, the number of message passing layers used in the forward propagation function corresponds to parameter k, the number of GNN layers (explained in concept 4), also known as the number of iterations. It can be seen in listing 4.5 that k=3. This means that for our graph, the aggregation will only reach as far as 3 hops away from the central nodes. If we wanted to extend or reduce this reach, we could add or take away layers from the *forward* method.

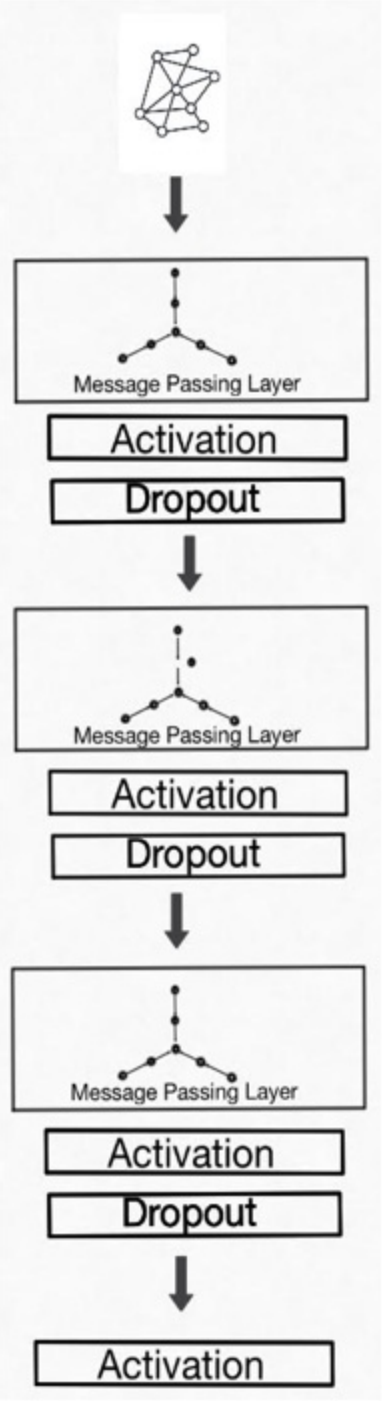
In the __init__ method, we define the message passing layers given the predefined GNN layers in listing 4.4. Other predefined GNN layers can be used or custom layers can be created and used. For these layers, the parameters are the input and output dimensions of the data. Though the 3 layers use the same GNN class, the input/output layers are distinct.

Listing 4.5 GraphSAGE full Architecture (using the GraphSage layer)

```
class GraphSAGE(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout
        super().__init__()
        self.dropout = dropout
        self.conv1 = SAGEConv(input_dim, hidden_dim) #A
        self.conv2 = SAGEConv(hidden_dim, hidden_dim) #A
        self.conv3 = SAGEConv(hidden_dim, output_dim) #A

    def forward(self, data):
        x = self.conv1(data.x, data.adj_t) #B
        x = F.Relu(x) #B
        x = F.dropout(x, p=self.dropout) #B

        x = self.conv2(x, data.adj_t)
        x = F.Relu(x)
        x = F.dropout(x, p=self.dropout)

        x = self.conv3(x, data.adj_t)
        x = F.Relu(x)
        x = F.dropout(x, p=self.dropout)

        return torch.log_softmax(x, dim=-1) #C
```

In addition, there are a few hyper-parameters related to the input and output sizes of the message passing, layer norm, linear layers, and the dropout. Figure 4.15 illustrates this architecture.

Figure 4.15 The model architecture of Listing 4.5.



4.3.3 Defining the Training Procedure

For the model training, we build routines for training and testing. It is common practice to implement testing and training via functions and classes. Below, we have functions for testing and training. Throughout the next few chapters we'll be defining multiple training and testing functions to build our GNN models. Again, it's important that you really understand these steps in order to understand the rest of this book.

For the *train* function, we specify the model object, the data object, node indexes for the training split, and the optimizer of choice. We call the *train* method on the model, then use the *zero_grad* method on the optimizer to zero the parameter gradients. We then do a forward pass of our data through the model, and use the resulting output to calculate the loss. Finally, we perform backpropagation with the *backward* method, and update the model weights with the *step* method.

```
def train(model, data, train_idx, optimizer):
    model.train()

    optimizer.zero_grad()
    out = model(data)[train_idx]
    loss = F.nll_loss(out, data.y.squeeze(1)[train_idx])
    loss.backward()
    optimizer.step()

    return loss.item()
```

For the *test* function, shown in listing 4.6, we can take advantage of OGB's built in evaluator to assess the model performance. We use its *eval* method to return the performance metric (for the ogbn-products dataset, the performance metric is accuracy).

From the ogb docs, the input format is:

```
input_dict = {"y_true": y_true, "y_pred": y_pred}
```

where *y_true* is the ground truth, and *y_pred* are the model predictions.

Within the *test* function, we:

- Run data through the model and get output.
- We then transform the output to get the classification predictions.

- With these predictions and the true labels, we use the evaluator to produce accuracy values for the train, validation, and test sets.

Listing 4.6 Test Routine for Model Training

```
def test(model, data, split_idx, evaluator):
    model.eval()

    out = model(data)
    y_pred = out.argmax(dim=-1, keepdim=True)

    train_acc = evaluator.eval({
        'y_true': data.y[split_idx['train']],
        'y_pred': y_pred[split_idx['train']],
    })['acc']

    valid_acc = evaluator.eval({
        'y_true': data.y[split_idx['valid']],
        'y_pred': y_pred[split_idx['valid']],
    })['acc']

    test_acc = evaluator.eval({
        'y_true': data.y[split_idx['test']],
        'y_pred': y_pred[split_idx['test']],
    })['acc']

    return train_acc, valid_acc, test_acc
```

4.3.4 Training

In the training routine, we call the train and test functions at each epoch to update the model parameters, and obtain the loss and the train/validation/test accuracies.

```
for epoch in range(1, 1 + epochs):
    loss = train(model, data, train_idx, optimizer)
    train_acc, valid_acc, test_acc = test(model, data, split_idx,
```

4.3.5 Saving the Model

As in Chapter 3, saving the model is a call to the *save* method of torch.

```
torch.save(model, '/PATH_TO_MODEL_DIRECTORY/model.pt')
```

4.3.6 Performing Inference

Outside of the training process, getting predictions with a model depends on making sure your data and model are on the same device (i.e., CPU, GPU, or TPU).

In the below example, we first have a set of data on the CPU. We need to move it to the model's device by using the *to* method and specifying the device. In the second line, the data is run through the model, with the output copied then placed back on the CPU.

```
data.to(device)
prediction = model(data).clone().cpu()
```

It's a common practice to set the device at the start of your session.

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
```

4.3.7 Applying GCN and GraphSage to Predicting Product Categories

4.4 Benchmarks to gauge performance

When training models and experimenting, having performance baselines and an idea of state of the art performance is a good way to understand how our models stack up against other solutions. Often, we can establish baselines for GNNs by using non-GNN solutions. For example, the product classification problem of this chapter can be first tackled by using a tree-based machine learning model. Another way to establish a baseline is to use pre-defined layers and the simplest architectures.

On the other side of the performance landscape are the state of the art solutions that push the limits of performance.

The GNN sector is relatively new, but there is a growing set of resources that allow one to benchmark performance against other GNN solutions for a limited set of domains and prediction tasks.

At the time of writing, two resources with such benchmarks are the Open Graph Benchmark site (https://ogb.stanford.edu/) and in the literature of research or conference papers. OBG features leaderboards for selected problem areas and datasets in those areas. If your problem involves node prediction, edge prediction, graph prediction, and involves a domain or dataset similar to the ones featured in the site (which include product networks, biological networks, molecular graphs, social networks, and knowledge graphs), examining the leaderboards for a particular task/dataset may be a good place to start.

Figure 4.16 shows a leaderboard from the Open Graph Database. In addition to test and validation performance, the GNN technique/layer is specified, as well as hardware details. Often links to papers and code are also available.

There are also a breadth of academic papers from journals, pre-prints, and conferences that can shed light on problem sets and domains that are not included in OGB's set.

Figure 4.16 Some performance values for node classification of the ogbn-products dataset, from the Open Graph Benchmark website. Each row consists of a solution architecture, its authors, performance values, and other information.

31	ClusterGCN+residual+3 layers	No	0.7971 ± 0.0042	0.9188 ± 0.0008	Horace He (Cornell)	Paper, Code	456,034	GeForce RTX 2080 (11GB GPU)	Oct 6, 2020
32	GAT with NeighborSampling	No	0.7945 ± 0.0059	Please tell us	Matthias Fey	Paper, Code	751,574	GeForce RTX 2080 (11GB GPU)	May 24, 2020
33	**GraphSAGE+FLAG**	No	0.7936 ± 0.0057	0.9205 ± 0.0007	Kezhi Kong	Paper, Code	206,895	GeForce RTX 2080 Ti (11GB GPU)	Oct 20, 2020
34	Cluster-GAT	No	0.7923 ± 0.0078	0.8985 ± 0.0022	Xiang Song	Paper, Code	1,540,848	EC2 P3.2xlarge (V100)	Aug 2, 2020
35	GraphSAINT (SAGE aggr)	No	0.7908 ± 0.0024	0.9162 ± 0.0008	Matthias Fey - OGB team	Paper, Code	206,895	GeForce RTX 2080 (11GB GPU)	Jun 10, 2020
36	ClusterGCN (SAGE aggr)	No	0.7897 ± 0.0033	0.9212 ± 0.0009	Matthias Fey - OGB team	Paper, Code	206,895	GeForce RTX 2080 (11GB GPU)	Jun 10, 2020
37	NeighborSampling (SAGE aggr)	No	0.7870 ± 0.0036	0.9170 ± 0.0009	Matthias Fey - OGB team	Paper, Code	206,895	GeForce RTX 2080 (11GB GPU)	Jun 10, 2020
38	Full-batch GraphSAGE	No	0.7850 ± 0.0014	0.9224 ± 0.0007	Matthias Fey - OGB team	Paper, Code	206,895	Quadro RTX 8000 (48GB GPU)	Jun 20, 2020
39	GraphSAGE	No	0.7829 ± 0.0016	Please tell us	Quan Gan (DGL Team)	Paper, Code	Please tell us	Please tell us	May 12, 2020
40	Full-batch GCN	No	0.7564 ± 0.0021	0.9200 ± 0.0003	Matthias Fey - OGB team	Paper, Code	103,727	Quadro RTX 8000 (48GB GPU)	Jun 20, 2020

4.5 Summary

- Graph Convolutional Networks and GraphSage are GNNs that use convolution, done by spatial and spectral methods, respectively.
- These GNNs can be used in supervised and semi-supervised learning problems; in this chapter, we applied them to the semi-supervised problem of predicting product categories.
- The Amazon Co-Purchasing Database, *ogbn-products*, consists of a set of products (nodes) linked by being purchased in the same transaction. Each product node has a set of features, including its product-category. This dataset is a popular benchmark for graph classification problems. We can also study how it was constructed to get insights on graph creation methodology.
- Convolution applied to graphs mirrors concepts used in deep learning. Two ways (among many) to approach convolution for graphs is via the ‘sliding window’ perspective, or a signal processing perspective. For the special qualities of graphs these methods must be adjusted.
- Convolutional GNNs fall roughly into two categories: spatial-based and spectral based. Spatial methods mainly consider the geometrical structure of a graph, while spectral methods are based on the eigenvalues of a graph's features.
- GCN, by Kipf, was the first GNN to take advantage of convolution. Its introduction has spawned a class of GCN-based architectures. It is distinguished by self-loop aggregation and symmetric normalization.
- GraphSage was a successful attempt to improve on Kipf’s GCN by introducing neighborhood sampling, and has in turn inspired architectures.
- A typical set of steps to train a node-prediction model is:
    - Preprocess - Prep the data
    - Define model - Define the architecture
    - Define training - Set of the training loop and learning criteria
    - Train - Execute the training process
    - Model Saving - Save the model
    - Inference - Use the saved model to make predictions
- There are many benchmarking resources for GNN-based inference. We suggest readers to build their own models and see how they compare!

4.6 References

GNN in Action Repository

https://github.com/keitabroadwater/gnns_in_action

Amazon Product Dataset

McAuley, Julian, Rahul Pandey, and Jure Leskovec. "Inferring networks of substitutable and complementary products." Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining. 2015.

GCN

Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016).

GraphSage

Hamilton, William L., Rex Ying, and Jure Leskovec. "Inductive representation learning on large graphs." In Proceedings of the 31st International Conference on Neural Information Processing Systems, pp. 1025-1035. 2017.

Deep Dive Into Convolutional Methods

Hamilton, William L. "Graph representation learning." Synthesis Lectures on Artificial Intelligence and Machine Learning 14.3 (2020): 51-89.

Other Illustrative Convolutional GNNs

Niepert, Mathias, Mohamed Ahmed, and Konstantin Kutzkov. "Learning convolutional neural networks for graphs." International conference on machine learning. PMLR, 2016.

Graph Signal Processing

Shuman, David I., et al. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains." IEEE signal processing magazine 30.3 (2013): 83-98.

5 Graph Attention Networks (GATs)

This chapter covers

· Introducing Attention as used in GNNs and how GATs are positioned in the GNN taxonomy

· Understanding attention and how is it applied to graph deep learning

· Implementing and applying GATs in a node-prediction problem

In the last chapter, we examined convolutional GNNs, focusing on GCN and GraphSage. In this chapter, we extend our discussion of convolutional GNNs by looking at a special variant of such models, the Graph Attention Network.

As with the previous chapter, our goal is to understand and apply Graph Attention Networks (GATs). While these GNNs use convolution as explained in chapter 4, they enhance this with an attention mechanism, that is, a mechanism which highlights important nodes in the learning process (vs. conventional convolutional GNNs which treat all nodes equally).

As with *convolution*, *attention* is a widely used method in deep learning outside of GNNs. Architectures that rely on *attention* (and *transformers*, a closely related method) have seen particular success in addressing natural language problems. As with the previous chapter, we'll provide some basic explanations of the larger underlying concepts, and connect them to GATs.

GATs shine when dealing with domains where some nodes have outstanding value or importance. Some examples of such domains are social networks, where some members of the network may have outsized importance in generating and spreading information; fraud detection, where a small set of actors and transactions perpetuate deception; and anomaly detection, where we find people, behavior, or events that fall outside the norm.

We will apply GATs to the domain of fraud detection, in particular the problem of detecting fake customer reviews. For this, we use a multi-relational network of user reviews derived from the YelpCHI dataset, which contains Yelp reviews for hotels and restaurants in the Chicago area.

This chapter is arranged as follows. Section 5.1 will introduce the example problem. Section 5.2 will cover background and theory of GATs. Section 5.3 will explain how a GAT solution will solve this problem and it is implemented in code. Code snippets will be used to explain the process, but the majority of code and annotation will be in the repo.

5.1 GATs Applied to Fraud

In this section, we will give some context of the fraudulent review problem, and describe the graph dataset to which we will apply our GAT, a collection of restaurant and hotel reviews from Yelp.

Overview of Fraud problem. **Fraud**, which we'll define as deception for some gain, crosses industries and domains. While fraud has been around since time immemorial, fraud in the digital domain seems to be increasing in impact and pervasiveness. US consumers lost $5.8B in fraud in 2021, an increase of 70% from the previous year. American businesses lose an average of 5% of their revenues to fraud; some of these losses end up getting passed to the consumer.

A subset of the fraud issue is deception in online product and service reviews. On consumer oriented websites and e-commerce platforms including Yelp, Amazon, and Google, it is common for ostensibly user-generated reviews and an aggregated rating to accompany the presentation and description of a product or a service.

In the US, more than 90% of adults trust and rely on these reviews and ratings when making a purchase decision. At the same time, many of these reviews are fake. Fakespot.com, a company that uses AI to detect fake reviews, estimated in 2021 that 30% of online reviews aren't real (Grill-Goodman).

One more level of nuance in this domain is that of **spam**. In general, spam is unsolicited or unwanted content that may or may not have fraudulent intent. For emails and SMS messages spam is familiar as unwanted messages. For user reviews, spam can be seen as useless reviews which are a waste of time to read. Such reviews could be written by an oblivious user, or by a bad actor that wants to saturate a product page to raise a rating.

Example Problem: Detect Spammy/Fraudulent User Reviews

Figure 5.1 Illustration of digital shopping with product categories.

Our Dataset: The Yelp (Multi-Relational) Spam Review

Spam or Fraudulent Review detection has been a well trod area in machine learning and natural language processing. As such, several datasets from the primary consumer sites and platforms are available. We will investigate review data from Yelp.com, a platform of user reviews and ratings that focuses on consumer services. On Yelp.com, users can look up local businesses in their proximity, and browse basic information about the business, as well as written feedback from users. Yelp itself filters reviews based on their internal measure or prediction of trustworthiness.

Explanation of problem

For this work, we'll use a multi-relational dataset (which I'll call Yelp Multirelational, or YelpMulti for short) that was derived from the YelpChi dataset (refs). YelpChi is a set of written reviews from Chicago-based Hotels and Restaurants. These reviews are classified depending on if they were filtered by Yelp's algorithm. YelpChi also contains information about the review writers, who are also labeled.

Yelp Chi - A set of labeled reviews with the corresponding labeled reviewers.	Yelp MultiRelational - Yelp-Chi processed into a multi-relational graph, with node features based on review text and user data.

# of Reviews: 67,395 13.23% filtered (fraudulent) reviews # of Reviewers: 38,063 20.33% spammers	# of Nodes (reviews): 45,954 14.5% filtered (fraudulent) nodes Node Features: 32 Total # of edges: 3,846,979 Types of Edges Same Reviewer: 49,315 Same Reviewed Business and Month: 73,616 Same Reviewed Business and Rating: 3,402,743

Figure 5.2 Simplified illustration of the multi-relational graph structure for the Multirelational Yelp Dataset. All nodes represent individual review with a label (spam, *S* or legitimate, *L*) and a feature vector. Connecting these nodes are three types of edges representing the relationship between the reviews. Black edges connect reviews from the same user; yellow edges connect reviews of the same business and same rating; and red edges connect reviews of the same business written in the same month

Figure 5.3 Node features (Rayana)

5.2 Understanding Attention and Its Use in GATs

In this section, we'll cover the basic concepts required to grasp graph

attention networks. We start with the more fundamental concepts of attention and transformers, focus on the GAT implementation introduced by Velickovic, then close with an overview of GAT variants that either build upon or use similar elements.

For further and deeper reading, references are provided at the chapter's end.

To summarize what is covered in this section, we cover:

· Attention and Transformers

· GATs as a variant of convolutional GNNs

Concept 1: Attention

The **attention** mechanism in machine learning is loosely analogous to human attention in everyday life. This concept refers to our ability to distinguish and isolate interesting parts of our sensory input and use these interesting parts to guide our ideas and decisions.

Attention allows us to block out the various noises in a crowded cafe and focus on a conversation with a friend in that same cafe. A parent can keep track of their toddler in a crowded playground by distinguishing them by look and sound. In a manhunt, bloodhounds can distinguish a particular scent to track a person across a vast forest.

In the machine learning version of attention, we want to distinguish and isolate data with characteristics that allow us to solve a learning problem. Problem domains with unstructured and inconsistent data (namely natural language and images) were the catalysts of attention based solutions. These solutions needed to deliver consistent and useful outputs from such data.

Natural Language Processing has seen a resounding success of attention methods. An example of a use case would be in language translation. Say we want to translate a story from English to Turkish. Before attention methods, the best ML solutions to this problem were limited by:

· Fixed Length Input Representations: Sentences in a story can have various

word lengths. However representations produced by early architectures only had one length. So, for example, this Grimm story, Goldilocks and the Three Bears has sentences as long as 20 words: "Now then, this must be Father Bear's chair, this will be Mother Bear's, and this one must belong to my friend, Baby Bear." It also has a sentence with one word: "Mm!". Both sentences would be represented using a vector of the same length. Say the length of the representative vector of 10. Expressing "Mm!" Using a vector of 10 may be adequate, but relatively speaking, using this same vector for the 20-word sentence would capture less meaning. In one case, we cram the meaning of 20 words into a vector of 10 versus 1 word into a vector of 10. [https://www.fairytales.biz/brothers-grimm/goldilocks-and-the-three-bears.html]

· Lack of Selective Weighings: Another problem of the representation described above is that there is no mechanism to identify important words in the sentences, and give these important words weight when constructing the output sentence. In the present example, for the 20-word sentence, 'then' would have as much importance as 'bear' in the representation.

· Weak on longer documents: This method works relatively well for small sentences, but has weaker performance on longer documents where words and sentences far apart in the document have some connection.

These problems were addressed by attention-based architectures and then transformer methods. Note that attention, like convolution, is a widely used concept in machine learning. Conceptually, there are several types of attention. Also, attention can be implemented using several algorithmic methods. Lastly, attention scores (values that indicate importance) can be computed differently.

Key attention-related concepts for GATs:

· Self Attention. This is a key method that enables transformers, and also key to the PyG implementations of GAT layers. In simple terms, self-attention yields attention scores (i.e., importance weights) of a set of nodes by calculating coefficients based on each pair of nodes in the set. These coefficient calculations use the node features. When we apply self-attention, we end up with a correlation matrix that will have information about the

relative importance of the input nodes.

· Multi-head Attention. What if we applied the self-attention process N-times, in parallel, with each separate process using some equal partition of the input data, but making the same calculations? This, in essence, is multi-headed attention. Simply using self-attention would be a head of 1. The advantages of using several heads versus one are an increase in training stability.

Figure 5.4 Conceptual Illustration of multi-head attention [Veličković].

Concept 2: GATs are a Variant of Convolutional GNNs

Though there are many ways to incorporate attention into an architecture, we will focus on two closely related models that apply the self-attention mechanism to convolutional GNNs, which were explained in chapter 4.

Recall that in a convolutional GNN, layers follow the following general process,

Layer: Filter(UPDATE-EMBEDDING + AGGREGATE-NODES) -> Activation Function -> Pooling

(5.1)

with different implementations using various methods to fulfill the elements above. GCN, for example, dispenses with UPDATE-EMBEDDING term, and calculates the AGGREGATE-NODES using self-loops and a special normalization (section 4.1, concept 5).

For a GCN, this aggregation step is based on the features of the neighboring nodes. All nodes in the neighborhood are treated equally, here. A GAT layer follows a similar aggregation, but enhances with aggregation with a self-attention mechanism. So, the resulting node features from passing through this layer will include the importance information of neighboring nodes as well as feature information.

Pytorch Geometric uses has two implementations of the convolutional GATs:

· GATConv. This is based on the Veličković paper. This architecture has been popular since its inception. Its application of self-attention, dubbed **static attention,** had some limitations, fixed by GATv2Conv. The limitation is that the importance ranking of nodes is across the entire graph, and not dependent on the updated node.

· GATv2Conv. This is based on the Brody paper. The authors fixed the limiting issue with GATConv, resulting in a layer that used **dynamic attention** (defined as attention that is closer to the broadly known idea of self-attention). The terms static and dynamic attention originated in this paper..

What are the tradeoffs of GATs versus the convolutional GNNs studied in chapter 4?

· Higher Performance. By being able to focus on the most important data during training, GATs in general have a higher performance than convolutional GNNs. Of course, this may not apply to specific domains and problems.

· Slower to Train. A tradeoff of higher performance is that GATs are slower to train.

· Scaling Issues. Related to the training slowness is the fact that the GAT implementations don't scale well. So, larger or more dense datasets will have a higher penalty from a memory and/or compute point of view.

5.3 Solving the Fraudulent Review Problem Using PyG

GNN interpretation of problem: A semi-supervised node classification problem

In this section, we will use GAT to perform node classification of the Yelp reviews, sifting the fraudulent from the legitimate reviews. First, we'll

establish baselines using non-GNN models: logistic regression, XGBoost, and sklearn's multilayer perceptron. Then we will apply PyG's GAT implementation to the problem. To summarize our process, we will:

· Load and Preprocess the dataset

· Define Baselines models and results

· Implement the GAT solution and compare to baseline results

5.3.1 Pre-processing

The dataset can be downloaded from Yingtong Dou's github repository (link in references), compressed in a zip file. The unzipped file will be in matlab format. Using the *loadmat* function from the *scipy* library, and a utility function from Dou's repository, we can produce the objects we need to start:

· A features object - containing the node features

· A labels object - containing the node labels

· and, an adjacency list object

Listing 5.1 Load Data

```
prefix = 'PATH_TO_MATLAB_FILE/'

data_file = loadmat(prefix +  'YelpChi.mat') #A

labels = data_file['label'].flatten() #B
features = data_file['features'].todense().A #B

yelp_homo = data_file['homo'] #C
sparse_to_adjlist(yelp_homo, prefix + 'yelp_homo_adjlists.pickle'
```

#A loadmat is a scipy function that loads matlab files

#B On these respective lines, we retrieve the node labels and features.

#C These lines retrieve and pickle an adjacency list. "Homo" means that this

adjacency list will be based on a homogenous set of edges, i.e., we get rid of the multi-relational nature of the edges.

Once the adjacency list is extracted and pickled, it can be called in the future using:

```
with open(prefix + 'yelp_homo_adjlists.pickle', 'rb') as file:

homogenous = pickle.load(file)
file.close()
```

Light Data Exploration

Let's examine some characteristics of our graph, including:

- The number of nodes (i.e., reviews)
- The number of node neatures
- Number and proportion of classes
- Distribution of the node features

Listing 5.2 Light Data Exploration

```
print(f'number of entries = {features.shape[0]}, number of featur

print(f'number of classes  = {np.unique(labels)}') #B
print(f'percentage false = {100*labels.sum()/len(labels):.2f}%')



print(f'features dtype = {features.dtype}, \n' +
      f'label dtype = {labels.dtype}') #C

fig,ax = plt.subplots(4,8,figsize=(15,15)) #D
for i in range(4):
    for j in range(8):
        ax[i,j].hist(features[:,8*i+j], 50)
```

#A Check the shape of the feature dataset.

#B Check number of classes and percentage attributed. The low percentage of false shows a high class imbalance.

#C Checking datatypes. Note current data is float64 and int64. Both will need to be translated into torch tensors when using pytorch geometric.

#D plot distribution of the node features.

These lines yield:

Table 5.1 Output of listing 5.2.

Item	Value
Number of Nodes	45954
Number of Features	32
Classes	0, 1 (2 classes)
Percentage of “false”-labeled nodes	14.53%
Data-types	Features = float64 Labels = int64

The feature distribution plot is below:

Figure 5.5 Distribution of node features generated using Listing 5.2.

5.3.2 Training Baseline Models

Next, we split our data into test and train sets, and apply 3 models for a baseline: logistic regression, XGBoost, and a multilayer perceptron.

First the test/train splitting:

```
from sklearn.model_selection import train_test_split

split = 0.2
xtrain, xtest, ytrain, ytest = train_test_split(features, labels,



print(f'Required shape is {int(len(features)*(1-split))}') #B
print(f'xtrain shape = {xtrain.shape}, xtest shape = {xtest.shape
print(f'Correct split = {int(len(features)*(1-split)) == xtrain.s
```

#A Split data into test and train sets, with a 80/20 split.

#B Double check the object shapes

We can use this split data for each of the 3 models. For this training, we are only using the node features and labels. There is no use of the graph data structure or geometry. In this sense, the 32 features are basically treated as a set of tabular data.

For the baseline models and for the GNNs, we'll use accuracy and ROC as performance metrics.

Logistic Regression

We use Sci-kit Learn's implementation of logistic regression, with the default hyperparameters.

```
from sklearn.linear_model import LogisticRegression

from sklearn.metrics import accuracy_score
from sklearn.metrics import roc_auc_score, f1_score

clf = LogisticRegression(random_state=0).fit(xtrain, ytrain) #A
ypred = clf.predict_proba(xtest)[:,1]
```

```
acc = roc_auc_score(ytest,ypred) #B

print(f"Model accuracy (logression) = {100*acc:.2f}%")
```

#A Logistic Regression model instantiation and training.

#B Accuracy score

This model yields an accuracy of 75.90%. For the ROC performance, we'll also use a function from Sklearn. We'll recycle the true positive rate and false positive rate to compare with our other baseline models.

```
from sklearn.metrics import roc_curve

fpr, tpr, _ = roc_curve(ytest,ypred) #A

plt.figure(1)
plt.plot([0, 1], [0, 1])
plt.plot(fpr, tpr)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.show()
```

#A ROC curve calculation, yielding false positive rate (fpr) and true positive rate (tpr

Figure 5.6 ROC curve for logistic regression baseline model.

XGBoost

The XGBoost baseline follows the logistic regression. We use a barebones model with the same training and test sets. For comparison, we differentiate the names of the generated predictions (named *pred2*), the true positive rate (*tpr2*), and the false positive rate (*fpr2*).

Listing 5.3 XGBoost Baseline and Plot

```
import xgboost as xgb
xgb_classifier = xgb.XGBClassifier()
```

```
xgb_classifier.fit(xtrain,ytrain)
ypred2 = xgb_classifier.predict_proba(xtest)[:,1] #A
acc = roc_auc_score(ytest,ypred2)

print(f"Model accuracy (XGBoost) = {100*acc:.2f}%")

fpr2, tpr2, _ = roc_curve(ytest,ypred2) #B

plt.figure(1)
plt.plot([0, 1], [0, 1])
plt.plot(fpr, tpr)
plt.plot(fpr2, tpr2) #B
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.show()
```

#A For comparison, we name the XGBoost predictions ‘ypred2’

#B For comparison, we distinguish the tpr and fpr of XGBoost and plot them alongside the logistic regression result

Figure 5.7 ROC curve for the XGBoost, shown with the logistic regression curve. We see that the XGBoost curve shows a better performance than the logistic regression.

XGBoost fares better than logistic regression with this data, yielding an accuracy of 89.25%, and with a superior ROC curve.

Multilayer Perceptron (MLP)

For the multilayer perceptron baseline, we use pytorch to build a simple, 3-layer model. As with pytorch, we establish the model using a class, defining the layers and the forward pass.

Listing 5.4 MLP Baseline and Plot

```
import torch #A

import torch.nn as nn
import torch.nn.functional as F
```

```
class MLP(nn.Module): #B
    def __init__(self, in_channels, out_channels, hidden_channels
        super(MLP, self).__init__()
        self.lin1 = nn.Linear(in_channels,hidden_channels[0])
        self.lin2 = nn.Linear(hidden_channels[0],hidden_channels[
        self.lin3 = nn.Linear(hidden_channels[1],out_channels)

    def forward(self, x):
        x = self.lin1(x)
        x = F.relu(x)
        x = self.lin2(x)
        x = F.relu(x)
        x = self.lin3(x)
        x = torch.sigmoid(x)

        return x

model = MLP(in_channels = features.shape[1], out_channels = 1) #C

print(model)

epochs = 100 #D
lr = 0.001
wd = 5e-4
n_classes = 2
n_samples = len(ytrain)

w= ytrain.sum()/(n_samples - ytrain.sum()) #E

optimizer = torch.optim.Adam(model.parameters(),lr=lr,weight_deca
criterion = torch.nn.BCELoss()

xtrain = torch.tensor(xtrain).float() #G
ytrain = torch.tensor(ytrain)

losses = []

for epoch in range(epochs): #H
    model.train()
    optimizer.zero_grad()
    output = model(xtrain)

    loss = criterion(output, ytrain.reshape(-1,1).float())
    loss.backward()
    losses.append(loss.item())

    ypred3 = model(torch.tensor(xtest,dtype=torch.float32))
```

```
    acc = roc_auc_score(ytest,ypred3.detach().numpy())
    print(f'Epoch {epoch} | Loss {loss.item():6.2f} | Accuracy =

    optimizer.step()

fpr, tpr, _ = roc_curve(ytest,ypred)
fpr3, tpr3, _ = roc_curve(ytest,ypred3.detach().numpy()) #I

plt.figure(1) #J
plt.plot([0, 1], [0, 1])
plt.plot(fpr, tpr)
plt.plot(fpr2, tpr2)
plt.plot(fpr3, tpr3)
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.show()
```

#A Import needed packages for this section

#B Define the MLP architecture using a class

#C Instantiate the defined model

#D Set key hyperparameters

#E Added to account for class imbalance

#F Define the optimizer and the training criterion

#G Convert training data to torch data types: torch tensors

#H The training loop, in this example, we have specified 100 epochs

#I For comparison we differentiate the tpr and fpr

#J Plotting all 3 ROC curves together.

Figure 5.8 ROC curves for all 3 baseline models. The curves for logistic regression and MLP overlap. The XGBoost model shows the best

performance for this metric.

The MLP run for 100 epochs yields an accuracy of 85.9%, in the middle of our baselines. Its ROC curve has a very similar profile to the logistic regression model's.

Table 5.2 Accuracy for the three baseline models.

Model	Accuracy
Logistic Regression	75.90%
XGBoost	89.25%
Multilayer Perceptron	85.93%

To summarize this section, we have run 3 baseline models, to use as benchmarks against our GAT solution. We didn't attempt to optimize these models, and ended up with XGBoost performing the best with an accuracy of 89.25%.

5.3.3 Training on GNNs: GCN and GAT

In this section, we will apply GNNs to our problem, starting with the GCN from chapter 4 and finally PyG's GAT implementation. There are differences in these two model implementations that preclude simply swapping one for the other. Alot of this has to do with the data preprocessing and dataloading. We'll discuss these differences as we cover the example.

Data Preprocessing

One critical first step is to prep the data for consumption in our training routine. This follows some of what has already been covered in chapters 2

and 4. Below, we do the following:

· Establish the Train/Test Split. We use the same *test_train_split* function from before, slightly tweaked to produce indices. And we only keep the resulting indices.

· Transform our dataset into pytorch geometric tensors. For this, we start with the homogenous adjacency list generated in the last section. Using NetworkX, we convert this to a NetworkX *graph* object. From there, we use the PyG *from_networkx* function to convert this to a PyG *data* object.

· Apply the train/test split to the converted data objects. Using the indices from the first step.

· Row-normalize the node feature vectors. So that for each node, features sum to 1.

Listing 5.5 Converting the datatypes of our training data

```
from torch_geometric.transforms import NormalizeFeatures

split = 0.2 #A
indices = np.arange(len(features)) #A
xtrain, xtest, ytrain, ytest, idxtrain, idxtest = train_test_spli




g = nx.Graph(homogenous) #B
print(f'Number of nodes: {g.number_of_nodes()}')
print(f'Number of edges: {g.number_of_edges()}')
print(f'Average node degree: {len(g.edges) / len(g.nodes):.2f}')
data = from_networkx(g) #B
data.x = torch.tensor(features).float() #B
data.y = torch.tensor(labels) #B
data.num_node_features = data.x.shape[-1] #B
data.num_classes = 1 #binary classification #B

A = set(range(len(labels))) #C
data.train_mask = torch.tensor([x in idxtrain for x in A]) #C
data.test_mask = torch.tensor([x in idxtest for x in A]) #C
```

```
transform = NormalizeFeatures() #D
transform(data) #D
```

#A Establish train/test split; we will only use the index variables.

#B Take the adjacency list and transform it into PyG data objects

#C Establish the train/test split in the data objects

#D Normalize node features

With the preprocessing done, we are ready to apply the GCN and GAT solutions.

GCN

We detailed the GCN architecture in chapter 4. In the listing below, we establish a 2 layer GCN, trained over 100 epochs.

One item of note is our use of the masks in our training. While we establish loss using the nodes in the training mask, for forward propagation, we must pass the entire graph through the model.

Listing 5.6 GCN Definition and Training

```
class GCN(torch.nn.Module): #A

def __init__(self, hidden_layers = 64):
        super().__init__()
        torch.manual_seed(2022)
        self.conv1 = GCNConv(data.num_node_features, hidden_layer
        self.conv2 = GCNConv(hidden_layers, 1)

    def forward(self, data):
        x, edge_index = data.x, data.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 torch.sigmoid(x)
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "c
print(device)
model = GCN()
model.to(device)
data.to(device)

lr = 0.01
epochs = 100

optimizer = torch.optim.Adam(model.parameters(), lr=lr, weight_de
criterion = torch.nn.BCELoss()

losses = []
for e in range(epochs): #C
  model.train()
  optimizer.zero_grad()
  out = model(data) #D

  loss = criterion(out[data.train_mask], data.y[data.train_mask].
  loss.backward()
  losses.append(loss.item())

  optimizer.step()

  ypred = model(data).clone().cpu()
  pred = data.y[data.test_mask].clone().cpu().detach().numpy()
  true = ypred[data.test_mask].detach().numpy()
  acc = roc_auc_score(pred,true)

  print(f'Epoch {e} | Loss {loss:6.2f} | Accuracy = {100*acc:6.3f
fpr, tpr, _ = roc_curve(pred,true) #E
```

#A Define a 2-layer GCN architecture

#B Instantiate model, and put model and data on the GPU.

#C Training loop.

#D Note: for each epoch, we feed the entire data object through the model, then we use the training mask to calculate loss.

#E False positive rate (fpr) and true positive rate (tpr) calculations

A training session of 100 epochs for the GCN yields an accuracy of 77.19%. So, better than the logistic regression, but not the other baseline models.

Table 5.3 Accuracy for the three baseline models.

Model	Accuracy
Logistic Regression	75.90%
XGBoost	89.25%
Multilayer Perceptron	85.93%
GCN	77.19%

GAT

Finally, we get to the GAT solutions. We'll apply the two PyG implementations (GAT and GATv2 explained earlier to our problem.

From an implementation point of view, one key difference between the previously studied convolutional models and our GAT models is the outsized GPU memory requirements of GAT models (see 'scaling issue' reference). The reason is that GAT requires a calculation of attention scores for every attention head and for every edge. This in turn requires pytorch's autograd to hold tensors in memory that scale up depending on the number of edges, the number of heads, and twice the number of node features.

To get around this problem, we divide our graph into batches and load these batches into the trainer. This is in contrast to what we did above with GCN where we trained on one batch (the entire graph). PyG's *NeighborLoader* (found in its *dataloader* module) allows such mini-batch training. NeighborLoader is based upon the paper "Inductive Representation Learning on Large Graphs" (see reference). Key input parameters for *NeighborLoader* are:

· *num_neighbors* - how many neighbor nodes will be sampled, multiplied by the number of iterations (i.e., GNN layers). In our example, we specify 1000 nodes over 2 iterations.

· *batch_size* - the number of nodes selected for each batch. In our example, we set the batch size at 128

Listing 5.7 Setting up NeighborLoader for GAT

```
from torch_geometric.loader import NeighborLoader

batch_size = 128
loader = NeighborLoader(
    data,
    num_neighbors=[1000]*2, #A
    batch_size=batch_size, #B
    input_nodes=data.train_mask,
)

sampled_data = next(iter(loader))
print(f'Checking that batch size is {batch_size}: {batch_size ==
print(f'Percentage fraud in batch: {100*sampled_data.y.sum()/len(
sampled_data
```

#A Sample 1000 neighbors for each node for 2 iterations

#B Use a batch size for sampling training nodes

In creating our GNN class with GAT, there are two key changes to make relative to our GCN-based classes. First, since we are training in batches, we want to apply a batchnorm layer. Secondly, we note that our GAT layers have an additional input parameter: *heads*, the number of multi-headed attentions. In our example, our first GATConv layer has 2 heads, specified in listing 5.9. The second GATConv layer has 1 head, since it is the output layer.

Listing 5.8 GAT based architecture

```
class GAT(torch.nn.Module):

def __init__(self, hidden_layers=32, heads=1, dropout_p=0.0):
        super().__init__()
        torch.manual_seed(2022)
        self.conv1 = GATConv(data.num_node_features, hidden_layer
```

```
        self.bn1 = nn.BatchNorm1d(hidden_layers*heads) #B
        self.conv2 = GATConv(hidden_layers * heads, 1, dropout=dr

    def forward(self, data, dropout_p=0.0):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = self.bn1(x) #B
        x = F.relu(x)
        x = F.dropout(x, training=self.training)
        x = self.conv2(x, edge_index)

        return torch.sigmoid(x)
```

#A GAT layers have ‘heads’ parameter. For this parameter, the layers preceding the output can have >1 for, while, the output layer would be 1.

#B Since we are performing mini-batch training, we add a batchnorm layer.

Our training routine for GAT is similar to the single batch GCN, except that a nested loop for each batch would contain the weight updates and loss calculations.

Listing 5.9 Training loop for GAT

```
lr = 0.01

epochs = 20

model = GAT(hidden_layers = 64,heads=2)
model.to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=lr,weight_dec
criterion = torch.nn.BCELoss()

losses = []
for e in range(epochs):
    epoch_loss = 0.
    for i, sampled_data in enumerate(loader): #A
        sampled_data.to(device)

        model.train()
        optimizer.zero_grad()
        out = model(sampled_data)
        loss = criterion(out[sampled_data.train_mask], sampled_da
        loss.backward()
```

```
            epoch_loss += loss.item()

            optimizer.step()

            ypred = model(sampled_data).clone().cpu()
            pred = sampled_data.y[sampled_data.test_mask].clone().cpu
            true = ypred[sampled_data.test_mask].detach().numpy()
            acc = roc_auc_score(pred,true)

        losses.append(epoch_loss/batch_size)

        print(f'Epoch {e} | Loss {epoch_loss:6.2f} | Accuracy = {100*
```

#A Nested loop for mini-batch training. Each iteration here is a batch of nodes loaded by NeighborLoader.

The steps outlined above would be the same for GATv2Conv, which can be found in the detailed code.

Training GATConv and GATv2Conv yields accuracies of 93.21% and 92.19%, respectively. So, our GAT models outperform the baseline models and GCN.

Table 5.4 Accuracy of the models.

Model	Accuracy
Logistic Regression	75.90%
XGBoost	89.25%
Multilayer Perceptron	85.93%
GCN	77.19%

GAT	93.21%
GATv2	92.19%

Figure 5.9 ROC curves for GCN and GATConv. The GATConv model shows the best performance for this metric.

5.4 Summary

· Graph Attention Networks (GATs) are a special class of convolutional GNNs that make use of the attention mechanism.

· Attention-based GNNs can be applied to conventional graph learning problems, but shine in anomaly and fraud detection problems.

· In this chapter, we applied them to the semi-supervised problem of classifying spam user reviews.

· The Yelp Multi-Relational Review dataset consists of reviews (nodes) connected by multiple types of edges representing common reviewers, dates written, or a commonly reviewed business. Each node has a label, and a set of features based on the review text and reviewer characteristics. Nodes in this dataset have been filtered as fraudulent or legitimate.

· GAT does not scale well with respect to GPU memory. To remedy this, doing mini-batch training is recommended.

5.5 References

Fraud

“Fake Review Fraud”, Alliance to Counter Crime Online, 2020, https://www.counteringcrime.org/fake-review-

fraud#:~:text=Fraudulent%20reviews%20constitute%20unfair%20or,includin

Grill-Goodman, Jamie, “Report: 30% of Online Customer Reviews Deemed Fake.” RISNews.com, 2021, https://risnews.com/report-30-online-customer-reviews-deemed-fake.

“Fraud costs the global economy over US$5 trillion”, Crowe.com, 2019. https://www.crowe.com/global/news/fraud-costs-the-global-economy-over-us$5-trillion

Iacurci, Greg, “Consumers lost $5.8 billion to fraud last year — up 70% over 2020”, cnbc.com, https://www.cnbc.com/2022/02/22/consumers-lost-5point8-billion-to-fraud-last-year-up-70percent-over-2020.html

Datasets

YelpChi Dataset: http://odds.cs.stonybrook.edu/yelpchi-dataset/

Yelp Multirelational Review Dataset: https://github.com/YingtongDou/CARE-GNN/blob/master/data/YelpChi.zip

Spaeagle method - Rayana, Shebuti, and Akoglu, Leman. "Collective opinion spam detection: Bridging review networks and metadata." In *Proceedings of the 21th acm sigkdd international conference on knowledge discovery and data mining*, pp. 985-994. 2015.

Attention and Transformers

Cristina, Stefania and Saeed, Mehreen. *Building Transformer Models with Attention*, 2022, MachineLearningMastery.com.

Allamar, Jay, “The Illustrated Transformer” http://jalammar.github.io/illustrated-transformer/

GAT

GAT - Veličković, Petar, Guillem Cucurull, Arantxa Casanova, Adriana

Romero, Pietro Lio, and Yoshua Bengio. "Graph attention networks." *arXiv preprint arXiv:1710.10903* (2017).

GATv2 - Brody, Shaked, Uri Alon, and Eran Yahav. "How attentive are graph attention networks?." *arXiv preprint arXiv:2105.14491* (2021).

Scaling Issue - *"GATConv costs huge gpu memory", Issue #527,* https://github.com/pyg-team/pytorch_geometric/issues/527

NeighborLoader - Hamilton, W., Ying, Z. and Leskovec, J., 2017. Inductive representation learning on large graphs. Advances in neural information processing systems, 30. https://arxiv.org/abs/1706.02216

BatchNorm - Cai, T., Luo, S., Xu, K., He, D., Liu, T.Y. and Wang, L., 2021, July. Graphnorm: A principled approach to accelerating graph neural network training. In International Conference on Machine Learning (pp. 1204-1215). PMLR. https://proceedings.mlr.press/v139/cai21e.html

Ma, Yao, Tang, Jiliang. Deep Learning on Graphs. Cambridge University Press, 2021.

6 Graph Autoencoders

This chapter covers

- Distinguishing between discriminative and generative models
- Applying AutoEncoders (AEs) and Variational AutoEncoders (VAEs) to graphs
- Building both types of Graph AutoEncoders (GAEs) with PyTorch Geometric
- Performing link prediction with GAEs

So far, we have covered how classical deep learning architectures can be extended to work on *graph-structured data*. In Chapter 4 we considered *convolutional* GNNs, which apply the convolutional operator to identify patterns within the data. In Chapter 5, we explored the *attention mechanism* and how this can be used to improve performance for graph-learning tasks such as node classification.

Both convolutional GNNs and attention GNNs are examples of *discriminative* models, as they learn to discriminate between different instances of data, such as whether a photo is of a cat or a dog. In this chapter, we introduce the topic of *generative models* and explore them through two of the most common architectures, Auto-Encoders (AEs) and Variational Auto-Encoders (VAEs). Generative models aim to learn the entire dataspace rather than separating boundaries *within* the dataspace, as do discriminative models. For example, a generative model learns how to generate images of cats and dogs (learning to reproduce a;; aspects of fur, face and form of a cat or dog, rather than learning just the features that separates two or more classes, such as the pointed ears and whiskers of a Siamese cat or the long floppy ears of a Cocker Spaniel. Let’s take a closer look at these two approaches.

As we shall discover, discriminative models learn to separate boundaries in dataspace while generative models learn to model the dataspace itself. By approximating the entire dataspace, we can sample from a generative model and create new and unseen examples of our training data. In our example

above, it means that we can use our generative model to make new images of a cat or dog, or even some hybrid version that has features of both. This is a very powerful tool and important knowledge for both beginner and established data scientists. In recent years, deep generative models, which are generative models that use artificial neural networks, have shown amazing ability in many language and vision tasks. For example, the family of DALL-E models are able to generate new images from text prompts while models like GPT-3 have been shown to mimic a whole range of general language tasks.

In this chapter, we'll be exploring a subclass of generative models, *deep* generative models. These are generative models that use artificial neural networks to learn the dataspace. We'll learn how to implement the autoencoder architecture and use this generative model to improve a classifier for handwritten digits. We'll also demonstrate how to generate entirely new examples of handwritten digits. We cover this in Section 6.1, before introducing two classical examples of generative models to classify digits in the MNIST dataset. We then demonstrate how these generative architectures can be extended to act on graph-structured data, leading to Graph Auto-Encoders (GAEs) and Variational Graph Auto-Encoders (VGAEs).

[...]

To demonstrate the power of generative approaches to learning tasks, we return to the Amazon Product Co-Purchaser Network introduced in Chapter 4. However, in Chapter 4, we learnt how to predict what category an item might belong to given its position in the network. In this chapter, we'll show how we can predict where an item should be placed in the network, given its description. This is known as *edge prediction* and comes up frequently, for example, when designing recommendation systems. We'll put our understanding of GAEs to work here to perform link prediction, building a model that can predict when nodes in a graph are connected.

In section 6.5, we briefly discuss some of other types of deep generative models and their extensions to graph-based learning. By the end of this chapter, you should know the basics of when and where to use generative models of graphs (rather than discriminative ones) and how to implement them when we need to.

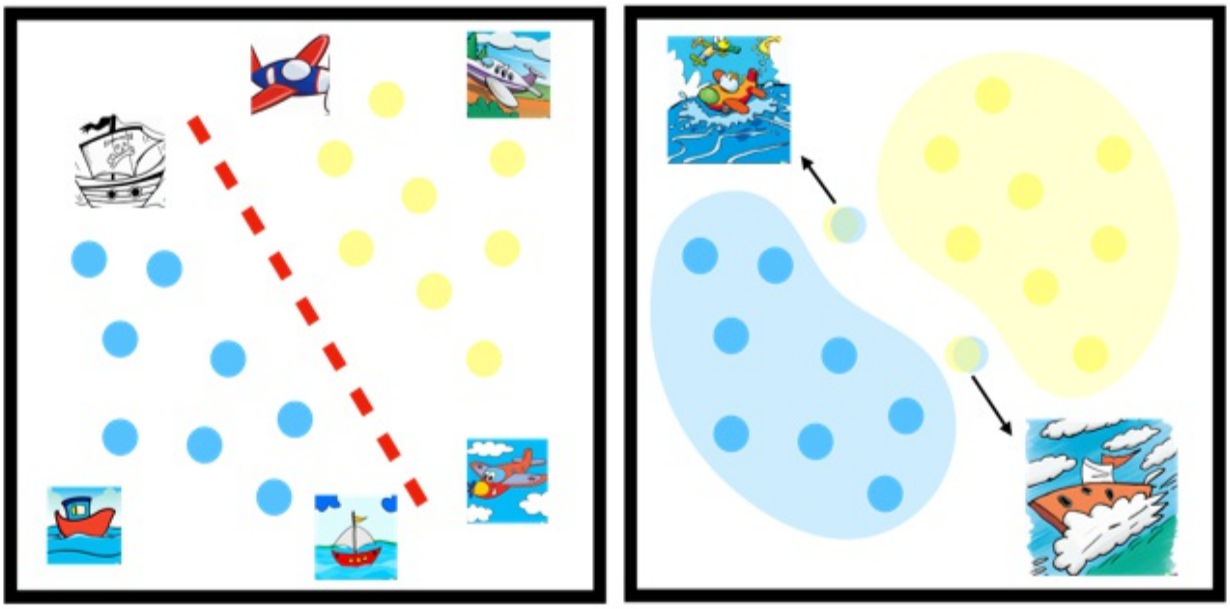
Note Where is the code?

Code from this chapter can be found in notebook form at the github repository and in Colab. Data from this chapter can be accessed in the same locations.

6.1 Generative models: Learning how to generate

A classic example of deep learning is, given a set of labeled images, how to *learn* what label to give to new and unseen images. If we consider the example of a set of images of boats and airplanes, we want our model to distinguish between these different images. If we then pass the model a new image then we want the model to correctly identify this model as, for example, a boat. An example of this is shown in Figure 6.1a. Discriminative models are models that learn to discriminate between classes based on their specific target labels. Both convolutional architectures, discussed in Chapter 4, and attention-based architectures, Chapter 5, are typically used to create discriminative models. However, as we shall see, they can also be incorporated into generative models. To understand this, we first have to understand the difference between discriminative and generative modeling approaches.

Figure 6.1 Comparison of generative vs discriminative tasks. On the panel on the left, we can see the discriminative model learn to separate out different images of boats and airplanes. On the right, the generative model attempts to learn the entire dataspace and this allows for new synthetic examples to be created such as a boat in the sky or an airplane on water.



As described in previous chapters, the original dataset that we use to train a model is referred to as our *training data,* and the labels that we seek to predict are our *training targets*. The unseen data is our *test data*, and we want to learn the *target labels* (from training) in order to classify the test data. Another way to describe this is using conditional probability. We want our models to return the probability of some target, Y, given an instance of data, X. We can write this as p(Y|X), where the vertical bar means that Y is 'conditioned' on X.

Generative models differ from discriminative models. As described, discriminative models learn to discriminate *between* classes. This is equivalent to learning separating boundaries of the data in the dataspace. However, generative models learn to model the dataspace itself. They capture the entire distribution of data in the dataspace and, when presented with a new example, they tell us how likely the new example is. Using the language of probability, we say that they model the *joint probability* between data and targets, P(X,Y). A typical example of a generative model might be a model that is used to predict the next word in a sentence (such as the auto-complete feature in many modern mobile phones). The generative model assigns a probability to each possible next word and returns those words that have the highest probability. Discriminative models tell you how likely the label for the instance is, without giving any information on the data itself.

Returning to our image-transport example, a generative model would learn the overall distribution of images. This can be seen in Figure 6.1b, where the generative model has learned where the points are positioned in the dataspace (rather than how they are separated). This means that generative models must also learn more complicated correlations in the data than their discriminative counterparts. For example, a generative model learns that “airplanes have wings”, “boats appear near water”. On the other hand, discriminative models just have to learn the difference between “boat” and “not boat”. They can do this by looking for tell-tale signs such as whether there is a mast, kneel or boon in the image. They can then largely ignore the rest of the image. As a result, generative models can be more computationally expensive to train. We give a first example of a generative model, the Auto-Encoder, in the next section and its extension to graph-structure data, the Graph Auto-Encoder.

Given discriminative models are computationally cheaper to train and more robust to outliers than generative models, we might wonder why we would want to apply a generative model to begin with. Generative models are efficient tools for when labeling data is relatively expensive but where generating datasets is easy to do. For example, generative models are increasingly being used in drug discovery where they generate new candidate drugs to test which match certain properties, such as the ability to reduce some disease. In a sense, generative models attempt to learn an underlying generative process that can recreate the synthetic data. This is contrasted to discriminative models that do not care about how the data was generated. Approximating how data is generated is a particularly powerful tool because it allows you to create new data instances. For example, none of the people shown in Figure 6.2 exist and were instead created by sampling from the dataspace, approximated using a generative model.

Figure 6.2 Figure showing synthetic faces

Synthetic examples created by generative models can be used to augment a dataset which is expensive to collect. Rather than taking lots of pictures of faces under every condition, we can use generative models to create new data examples (such as a person wearing a hat, glasses, and a mask) to increase our dataset to contain tricky edge cases. These synthetic examples can then be used to further improve our other models (such as one that identifies when someone is wearing a mask).

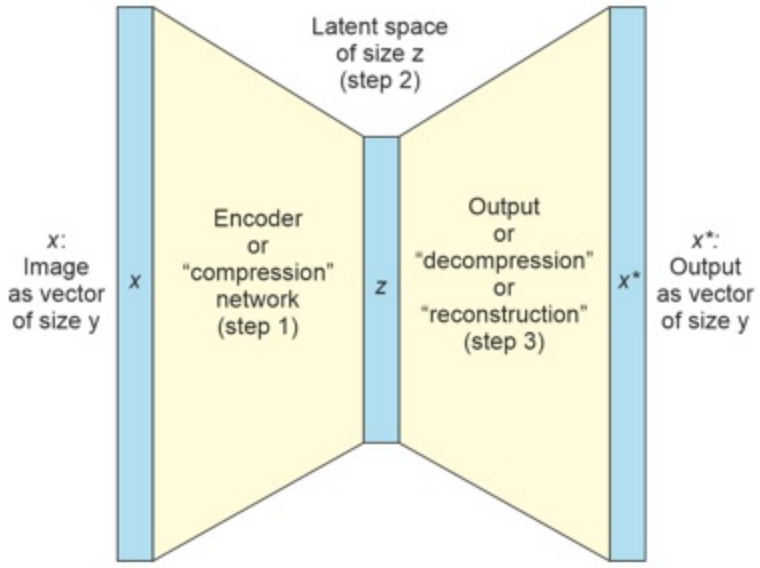
In fact, discriminative models are often used downstream of geneartive models. This is because generative models are typically trained in a 'self-supervised' way, without relying on data labels. They learn to compress (or *encode)* complex high-dimensional data to lower dimensions and these low-dimensional representations can be used to better tease out underlying patterns within our data. This is known as dimension reduction and can be very helpful in clustering data or in classification tasks. Later we will meet the MNIST dataset, which contains handwritten digits. We will see how generative models can separate these images into their respective digits without ever seeing their labels. In cases where annotating each datapoint is expensive, generative models can be huge costsavers. In the next few sections we'll discuss the autoencoder structure, one of the most popular deep generative models and we'll see exactly how generative models can allow us

to do more with less.

6.2 Deep generative model: Autoencoders

Deep generative models are generative models which use artificial neural networks to model the dataspace. One of the classic examples of a deep generative model is the autoencoder. Autoencoders contain two key components, the encoder and the decoder, both represented by neural networks. They learn how to take data and encode (compress) it into a low dimensional representation as well as decode (uncompress) it again. Figure 6.3 shows a basic autoencoder taking a large image as input and compressing it (step 1). This results in the low dimensional representation, or latent space (step 2). The autoencoder then the reconstructed the image (step 3) and the process is repeated until the reconstruction error between input image (x) an output image (x*) is as small as possible.

Figure 6.3 Structure of an autoencoder - taken from "GANs in Action"



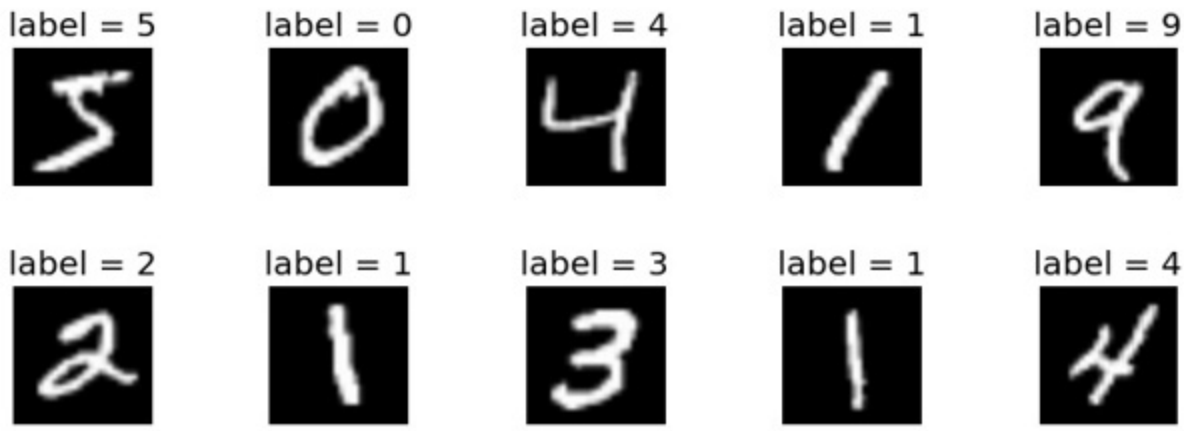
In order to learn more about autoencoders and how they work, we will apply them to one of the most famous datasets in machine learning, the MNIST digits dataset.

6.2.1 Applying a discriminative model to the MNIST dataset

Imagine you are helping a business that has no electronic records at all. Instead, everything from receipts to stock orders from the last ten years has been handwritten and stored on scraps of paper. The business owner wants you to convert all of these records to electronic versions. To do this, you first take pictures of all of the paper records. Next, you need to extract the handwritten digits and numbers in these photos to electronic text. To achieve this, you will almost certainly end up using the MNIST dataset as a benchmark. MNIST is a handy tool wherever we want to benchmark classification tasks in computer vision. It has also become one of the de facto datasets for introducing deep generative models. We'll follow the same tradition here and then go on to see how we can extend deep generative models to work on graph-base data.

Originally part of a much larger dataset of handwritten letters and numbers, the MNIST dataset consists of handwritten digits centered in a fixed-size image (28 x 28 pixels). It contains a training set of 60,000 examples and a test set of 10,000 examples, with ten classes (digits 0 - 9). We will use the MNIST dataset to build a discriminative model that can separate out different digits based on their image. This is a classification problem, we want to predict the numerical digit given an image of that digit. We'll then show how generative models can improve the performance of our classifier to help out our paper-loving business owner. We start with discriminative models to introduce the idea of dimension reduction for classification tasks before explaining how autoencoders are also performing a type of dimension reduction that can be used to improve discriminative models in just the same way.

Figure 6.4 MNIST dataset: -taken from a google image search. Will replace with code-generated example



The code in Listing 1.1 shows us how to load the MNIST dataset. Here, we're using torchvision to load and save our data. This is useful as torchvision separates test and training sets as well as our feature data (images) from our target labels. We are also transforming the data into tensors so we can use it later with pyTorch.

Listing 6.1 Loading the MNIST dataset

```
from torchvision import datasets    #A
from torchvision.transforms import ToTensor    #A

train_dataset = datasets.MNIST(root = 'datasets', \    #B
                               train = True, \    #B
                               transform = ToTensor(),\    #B
                               download = True)    #B

test_dataset = datasets.MNIST(root = 'datasets', \    #C
                              train = False, \    #C
                              transform = ToTensor(),\    #C
                              download = True)    #C
```

Having loaded the data, we next train a simple classifier on the dataset. Here, we're going to be using a random forest (RF) classifier but many others could also be tried. We just want a benchmark case to test against.

Listing 6.2 Base case classifier

```
from sklearn.ensemble import RandomForestClassifier    #A
from sklearn.metrics import accuracy_score    #A
```

```
seed = 50    #F


X_train = train_dataset.data.numpy().reshape(-1,28*28)    #B
y_train = train_dataset.targets.numpy()    #B

X_test = test_dataset.data.numpy().reshape(-1,28*28)    #B
y_test = test_dataset.targets.numpy()    #B

clf = RandomForestClassifier(max_depth=2, random_state=seed)    #
clf.fit(X_train, y_train)    #C
y_pred = clf.predict(X_test)    #D
print(accuracy_score(y_test, y_pred))    #E
```

We now have our benchmark accuracy for the MNIST dataset, which is just over 60%. To get this, we flattened each image into a vector of size 784, which is the pixel height (28) times the pixel width (28) of each image, along with the target labels. In the next few sections, we will investigate how we can reduce the dimensions of our data from a vector of length 784 to one of 10 and how this can end up improving the accuracy of our classifiers. We will show how we can get similarly high performance using fewer test samples which means fewer annotations needed. This is a great help in saving time and costs, and will hopefully make the paper-loving business owner happy!

6.2.2 Applying PCA to the MNIST dataset

So far, we developed a discriminative model that learns to separate out data based on given labels. We next investigate how to reduce the dimensionality of our data and, in some cases, improve on a classifier trained on all our data. We note that reducing the number of dimensions doesn’t always mean better performance for our classifier as, if we reduce dimensions too much then we lose important information and our classifier accuracy will worsen. We’ll explore some of these effects in what follows.

First, we're going to build a generative model to help us classify these digits. The first generative model we meet, the autoencoder, learns discrete estimates of the dataspace and, as we'll see, is also able to create entirely new digits from scratch. Once we understand the basics of autoencoders, we'll apply this method to graphs (graph autoencoders), before learning about the

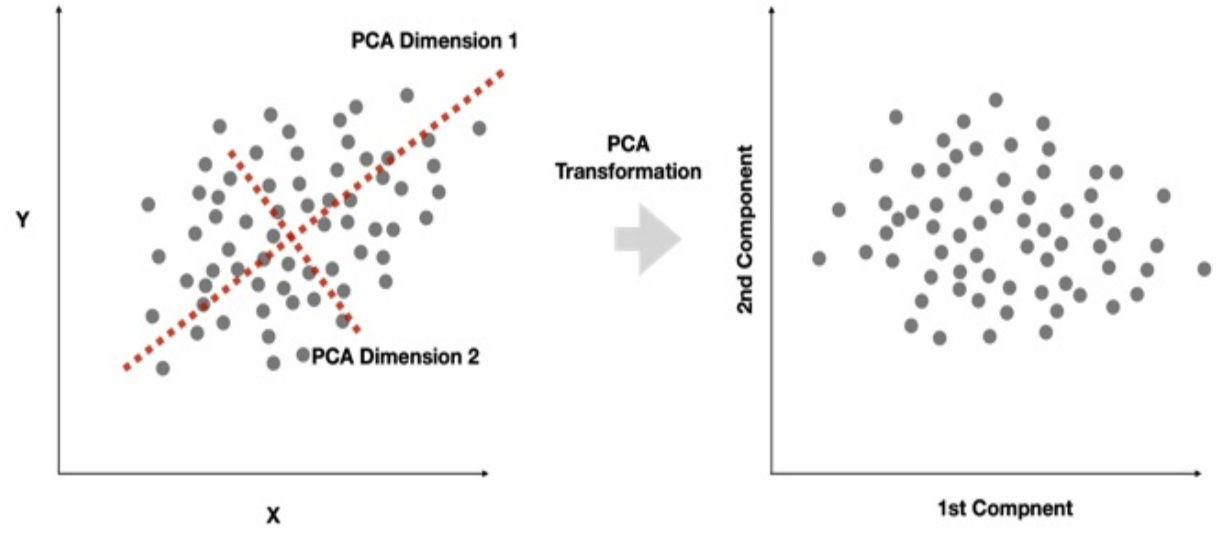
closely related variational autoencoder, which models the entire distribution of data as a probability density rather than point estimates. However, before we consider these generative models, let's first revisit the idea of dimension reduction using one of the most well-known methods in data science, principal component analysis or PCA. For more information about principal component analysis, with a deep dive into the theory, check out [Math and Architectures of Deep Learning - Manning Press].

Principal component analysis as linear dimension reduction

Fundamental to understanding autoencoders is the idea of dimension reduction. Dimension reduction is the concept that we can project or compress our dataspace on lower dimension spaces. This is exactly what we do when applying PCA, one of the most commonly deployed machine learning techniques. It is typically used to compress the data as a pre-processing step, identifying or creating new features that are then passed to other models. Here, we'll use PCA to reduce the dimension of each image and pass this to our RF classifier.

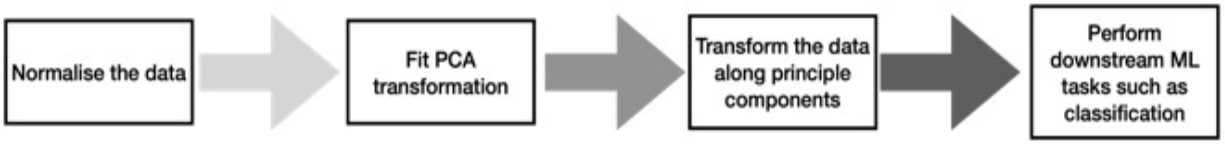
PCA takes an initial dataset and uses a linear transformation to project the data onto a subspace (with reduced dimension). Linearity is important here as we deploy methods from linear algebra to transform the data. Specifically, we look at the principal components of the covariance matrix of the data, as represented by the eigenvalues and eigenvectors, and use these to create a transformation along which we represent the majority of variance in our data.

Figure 6.5 Schematic of PCA. We calculate the variance using the covariance and then transform the data into a linear space that describes the majority of the variance in each component.



The transformation of the data gives new features which can then be used for downstream learning tasks or directly on the data to cluster and classify. This is shown in the process diagram in Figure 6.6. The first step is to apply PCA to our dataset, which in our case is the MNIST dataset. The code for this is given in Listing 6.3.

Figure 6.6 Process diagram for using PCA to perform dimension reduction

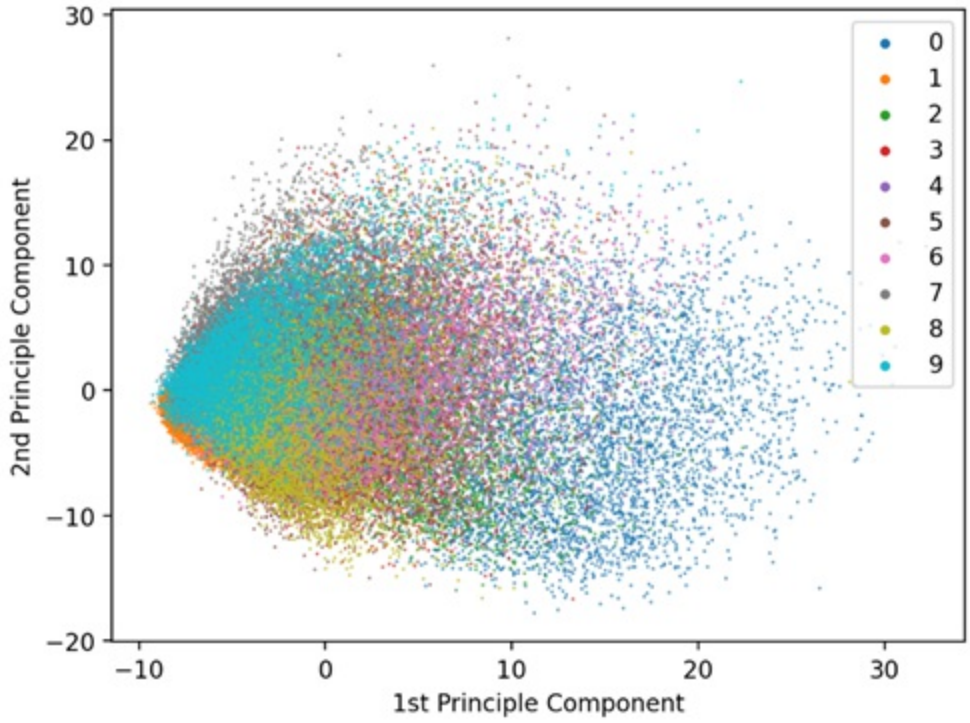


Listing 6.3 PCA applied to the MNIST dataset

```
from sklearn.decomposition import PCA    #A
from sklearn.preprocessing import StandardScaler    #B

X = train_dataset.data.numpy()
labels = train_dataset.targets.numpy()

X = X.reshape(X.shape[0],28*28)    #C
X = StandardScaler().fit_transform(X)    #C
pca = PCA(n_components=2)     #E
Xpca = pca.fit_transform(X)    #D
```

In Figure 6.7, we can see an example of the two components, and how plotting all the data points with labels in this new transformed coordinate space allows us to separate out some of the different digit classes. This type of visualisation is an intuitive way to understand how a dimension reduction technique is working. As we separate out the classes, we make our classifiers' job easier and can expect better accuracy. We can see that PCA doesn't do a great job here, as it hasn't separated out the different digits very much at all.

Figure 6.7 Example of PCA applied to the MNIST dataset.



We can now pass our newly transformed dataset to our classifier. Specifically, we first take our dataset (as images) and flatten each image before transforming the data into a low dimensional representation using PCA. We then pass this newly transformed data, which has the same number of dimensions as the number of components in our PCA, to our classifier. This is described in Listing 6.4.

Listing 6.4 Classifier using PCA transformed data

```
X_train = test_dataset.data.numpy().reshape(-1,28*28)
```

```
X_train = StandardScaler().fit_transform(X_test)    #A
X_train = pca.fit_transform(X_test)    #A
y_train = train_dataset.targets.numpy()


X_test = test_dataset.data.numpy().reshape(-1,28*28)
X_test = StandardScaler().fit_transform(X_test)    #A
X_test = pca.fit_transform(X_test)    #A
y_test = test_dataset.targets.numpy()

clf = RandomForestClassifier(max_depth=4, random_state=0)    #B
clf.fit(X_train, y_train)    #B
y_pred = clf.predict(X_test)    #B

print(accuracy_score(y_test, y_pred))    #B
```

When we pass the PCA transformed data to our RF classifier, the RF classifier accuracy falls to just over 50%. This is because PCA is typically used with more than two components, where the number of components are chosen based on the amount of variance they describe. For example, when we increase the number of components to 10, the accuracy increases to around 55%. We can investigate the amount of variance described by each component by plotting the eigenvectors from largest to smallest (also known as a scree plot). As we'll later see, deep generative models also require a choice of how many dimensions to reduce to but without the interpretability of PCA. Instead, the number of dimensions becomes a hyper-parameter to tune, similar to the learning rate or decay rate of a neural network. In general, size matters for dimension reduction. The central idea is that we can use transformations to reduce the dimension of our data. While this hasn't improved model performance, yet, this is the same concept that we'll be using when we train and deploy our autoencoders later in the chapter.

As discussed, autoencoders also perform dimension reduction. However, unlike PCA, autoencoders do not have the same linearity constraints. Instead, autoencoders use artificial neural networks to learn a non-linear mapping between datasets and a low-dimensional space. This gives more flexibility in how we encode our data. In the case of generative models, this low-dimensional space is commonly referred to as the 'latent space' or the 'bottleneck'. The intuition becomes clear when we use the phrase bottleneck, an autoencoder learns how to squeeze the dataspace into a low-dimensional space, funneling the data through a bottleneck. When we train our model, we

can then look at how the data has been restructured in the latent space and then perform clustering or classification, just as above in Listing 6.4.

The dimension reduction step reflects the 'encoder' part of an auto-encoder. But in order to construct the encoder, and the corresponding latent space, we also need to have a metric for how well the encoder is preserving the structure of the data. As an example, we don't want our encoder to map images of '2' digits to the class of '9' digits. In many learning methods, we ensure that data is correctly paired with labels during the training process. However, autoencoders (and many other generative methods) are examples of unsupervised learning. They do not use or need labels for training. This is the ‘auto-’ part of an autoencoder.

6.2.3 Applying a generative model to the MNIST data

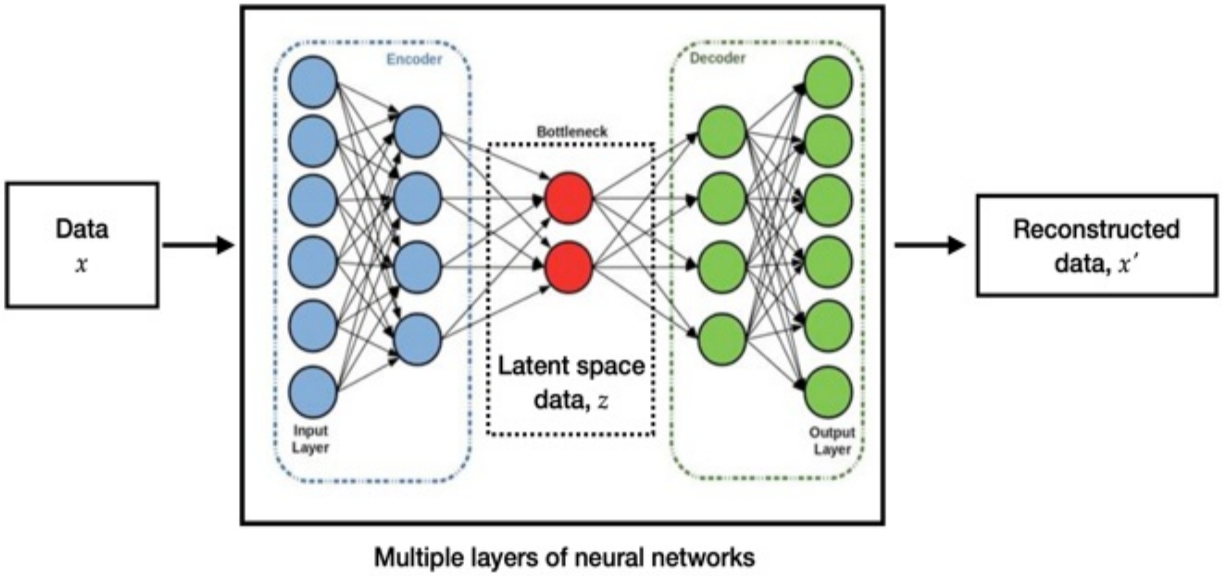
So far, we have shown how dimension reduction methods can be used before classification tasks. In the context of our paper-to-digital records problem, this means transforming each photo into a lower dimensional space and then using a classifier. We have also learnt that autoencoders perform a type of non-linear dimension reduction and they do this in a fully unsupervised way (without access to training labels). We will shortly be applying the autoencoder model to the MNIST dataset to see how this can further help us help the paper-loving business owner. Before that, we need to understand how autoencoders learn a nonlinear transformation, from data to latent space.

Autoencoders achieve this by constructing an encoder that projects data to the latent space and a decoder which maps the latent space back to the original dataset. This is essentially an inverse transformation of the encoder but, as the encoder is nonlinear, is also a nonlinear function and is trained using artificial neural networks. Autoencoders train both encoder and decoder simultaneously, by minimizing the difference between the original dataset and the data that has passed through the autoencoder.

We can see the structure of an autoencoder in Figure 6.8. The data, x, passes through a series of neural network layers that represent the encoder. These layers are progressively smaller, contractive, until the data is transformed into the low dimension version, z. This low dimensional output space is the latent

space. Finally, a series of neural network layers sequentially increases the dimension from the latent space back to the original dataspace dimensions. These neural network layers are the decoder. Data that passes from latent space through the decoder is the reconstructed data, x'. Autoencoders learn both encoders and decoders by minimising the difference between original data and the reconstructed data. This is achieved by minimising the reconstruction loss, x - x'.

Figure 6.8 Structure of an autoencoder. Input data is passed to the input layer of the neural networks and is compressed/encoded by smaller subsequent layers until there is a low dimensional representation known as the latent space. These are then decompressed/decoded by the decoder layers to reconstruct the data. (the central image is taken from a google image search and will need to be redrawn).



In the next few sections, we'll build our own autoencoder to apply to the MNIST dataset and use this to further improve on the accuracy of our classifier. Here, we are motivated by the businessman who wants to classify handwritten texts in his store. We want to get the best model we can that allows us to classify the digits. It would also be helpful to be able to investigate the data and identify hard to classify examples. Once we have built and trained our autoencoder of the digits, we'll look at the latent space and see how this is done. We will see that the latent space allows us to separate out the digits better than when using PCA. We'll also see how we can sample from the latent space to generate new images, demonstrating how

autoencoders are a class of *generative models.*

Building an autoencoder

We're next going to be building an autoencoder that we can use to improve our RF classifier. By using an autoencoder, we can create more powerful latent representations and, hence, improve the accuracy of our classifier. We will implement this architecture in PyTorch and apply it to the MNIST dataset. In the code below, we can see the encoder layers and the decoder layers, which match the structure depicted in Figure 6.6.

Listing 6.5 Encoder & Decoder Layers

```
import torch.nn.functional as F    #A
import torch.nn as nn    #A

class Encoder(nn.Module):    #B
    def __init__(self, input_size, layers, latent_dims):    #B
      super().__init__()    #B
      self.layer0 = nn.Linear(input_size, layers[0])    #B
      self.layer1 = nn.Linear(layers[0], layers[1])    #B
      self.layer2 = nn.Linear(layers[1], latent_dims)    #B

    def forward(self, X):    #C
      X = torch.flatten(X, start_dim=1)    #C
      X = F.relu(self.layer0(X))    #C
      X = F.relu(self.layer1(X))    #C
      return self.layer2(X)    #C

class Decoder(nn.Module):
    def __init__(self, latent_dims, layers, output_size):    #D
      super().__init__()    #D
      self.layer0 = nn.Linear(latent_dims, layers[1])    #D
      self.layer1 = nn.Linear(layers[1], layers[0])    #D
      self.layer2 = nn.Linear(layers[0], output_size)    #D

    def forward(self, z):    #E
      z = F.relu(self.layer0(z))    #E
      z = F.relu(self.layer1(z))    #E
      z = torch.sigmoid(self.layer2(z))    #E
      return  z #E
```

Here, we can see the structure of the autoencoder in terms of the encoder and

decoder networks. The encoder is first passed the original size of the data as input size. Then each layer shrinks the data size until it matches the target dimension of the latent space. Later on, we'll be defining the layers to be of size 512 and 256 while the dimension of the latent space will be 2. The layers reduce to represent the bottleneck nature of an autoencoder. Additionally, the latent space dimension matches the number of components in our PCA transformation in Listing 6.3. The shape and size of these layers are typically tuned to improved model performance.

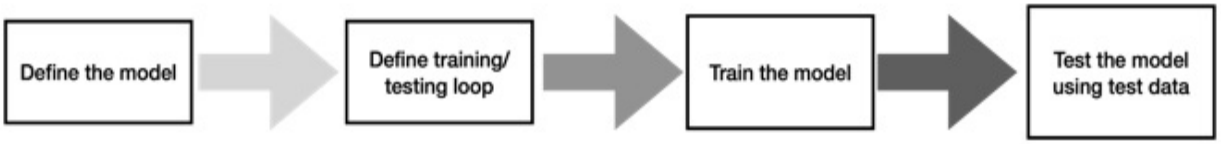
Next we see the decoder network. The decoder shares the same structure as the encoder but in reverse. First the decoder network takes the data from the latent space and increases it's dimensions to match the last layer of the encoder. It then continues to increase the dimensionality until it matches the input size of the encoder. Hence, the input and output size for both networks will be equal.

Listing 6.6 Autoencoder structure

```
class Autoencoder(nn.Module):    #A
    def __init__(self, input_size, layers, latent_dims):    #A
      super().__init__()    #A
     self.encoder = Encoder(input_size, layers, latent_dims)    #
     self.decoder = Decoder(latent_dims, layers, input_size)    #

    def forward(self, x):    #B
      z = self.encoder(x)    #B
      return self.decoder(z)    #B
```

Both of these are combined in the Autoencoder class above. Note that, as discussed above, the output for the decoder is the input size for the encoder. By breaking up the encoder and decoder networks, we are able to access the encoder part of the model directly post training. This is helpful for when we want to directly encode new data. Next, we define training and evaluation functions to train and evaluate our model. These functions are relatively boilerplate versions of other train and evaluate functions we used in earlier chapters. We repeat the overall process in Figure 6.9.

Figure 6.9 Standard process for training a deep learning model that we will be following throughout this chapter.



Listing 6.7 Preparing for training

```
input_size, latent_dims = 28*28, 2     #A
layers = [512, 256]    #B

model = Autoencoder(input_size, layers, latent_dims)    #C

criterion = nn.MSELoss()    #D
optimizer = torch.optim.Adam(model.parameters(), lr= 0.001)    #D
```

Above we specify the network dimensions, including input size, hidden layer sizes, and the latent dimensions. We will first use a very low dimensional latent space to encode our model, with only 2 dimensions. This is to compare our nonlinear transformation from the autoencoder to the one generated by PCA earlier in the chapter. Note, we have not passed any reference to the labels here. This network will be trained using the data only. Here we use the mean square error loss as our reconstruction loss, which will be a measure of how `different' the output image is from the passed input image.

Listing 6.8 Training function

```
def train(model, iterator, criterion, optimizer):

    model.train()    #A
    epoch_loss = 0    #B

    for x,_ in iterator:
        optimizer.zero_grad()    #C
        xbar = model(x)    #D
        xbar = xbar.reshape((-1, 1, 28, 28))    #E

        loss = criterion(xbar, x)    #F
        loss.backward()    #G

        optimizer.step()    #H
        epoch_loss += loss.item()    #I

   print(f"training loss: {epoch_loss /len(iterator):3.7}")
```

This describes our training function, a relatively standard format for training artificial neural network models. The only unusual step is that we reshape the data back to an image after it is passed to the decoder. Later in the chapter, we will reference only the specific parts of this training function that we alter in order to turn this autoencoder model to a variational autoencoder. Here, we emphasize that we are aiming for a small reconstruction error between our original image and that reconstructed by the autoencoder.

Listing 6.9 Evaluation function

```
@torch.no_grad()    #A
def evaluate(model, iterator, criterion, optimizer):
    model.eval()    #B

    epoch_loss = 0

    for x,_ in iterator:
        xbar = model(x)    #C
        xbar = xbar.reshape((-1, 1, 28, 28))

        loss = criterion(xbar, x)    #D
        epoch_loss += loss.item()    #E

    print(f"test loss: {epoch_loss /len(iterator):3.7}")
```

This is the evaluation function which we will run on the test data. This is again a generic function for evaluating a neural network.

Listing 6.10 Train & Test loop

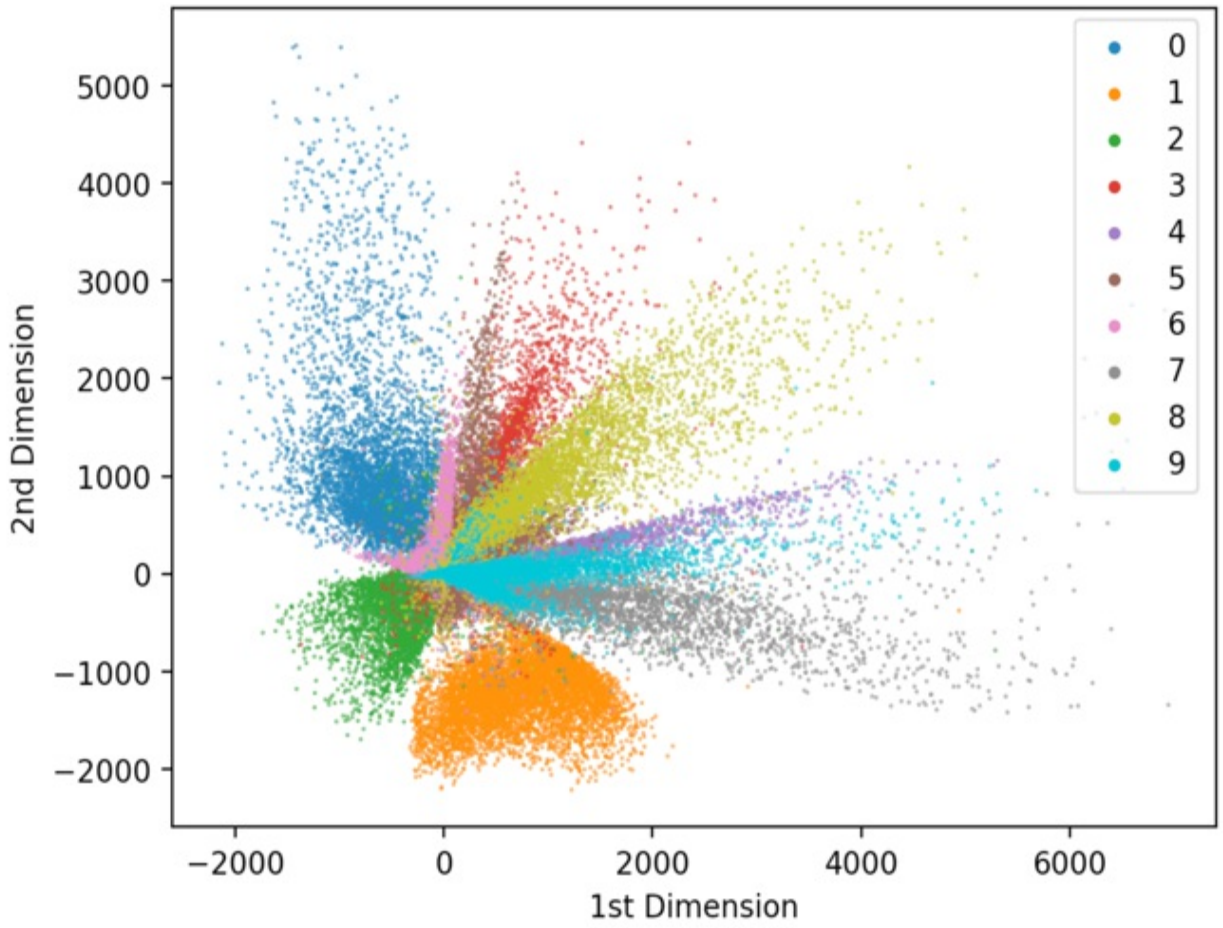
```
epochs = 20    #A
for e in range(1, epochs+1):    #B
  print(f'epoch = {e}')    #B
  train(loaders['train'], model=autoEncoder, criterion=criterion)

evaluate(loaders['test'], model=autoEncoder, criterion=criterion)
```

Having defined our model, as well as training and testing loop, we next train our model of 20 epochs and test the model on the unseen test data. All this code and training and test output as well as performance is given in our Github repo (https://github.com/keitabroadwater/gnns_in_action). We next turn to examine how well our model has learnt to represent the dataset. We

will first look at how well our model learns to represent and separate the MNIST data. We then will use the reduced dimensional data with our classifier and test to see how it impacts performance.

The accuracy of the autoencoder, given in Listing 6.7, represents how well the autoencoder has learnt to reproduce the data. Essentially, it is a measure of how well the generative model is learning to model the dataspace. If we plot our latent space, colored by class labels, we can see that the model has correctly learnt to separate out the different digits. This is shown in Figure 6.10.

Figure 6.10 Autoencoder applied to the MNIST data. The first two dimensions are able to separate out the different digits much better than the linear transformation performed by PCA.



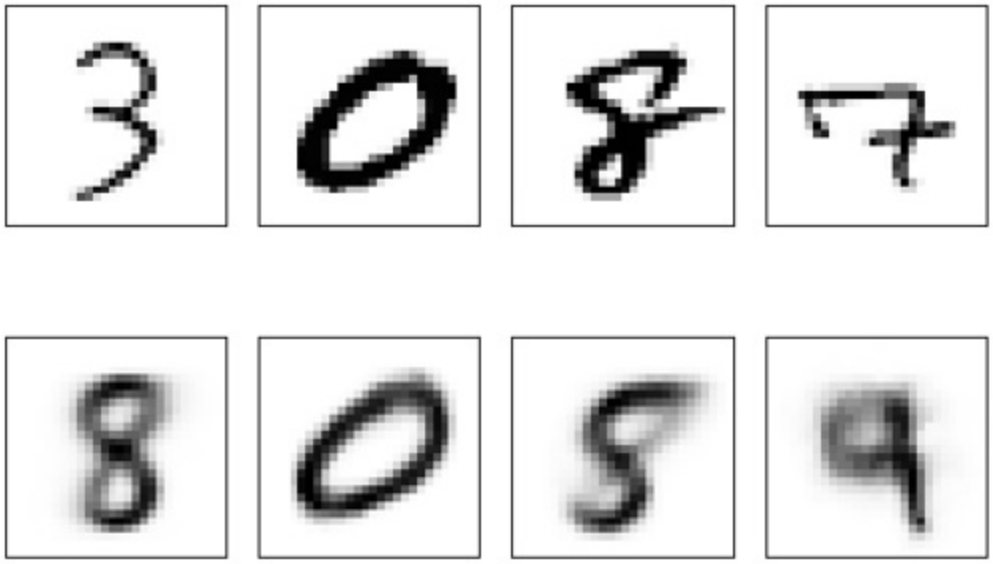
To generate this image, we remove the decoder part of our network and just examine the encoded data. As we have specified that the latent dimension is

dimension 2, we can plot all of the data in a single 2D figure. When we color based on our labels, we see that the different digits are now much better separated than when we applied the PCA transform. This is despite the fact that we reduced the dimensionality by the same amount. One of the main strengths here is that our model was trained without using any of the original class labels. It learnt to separate the data using only the data, and without access to any extra information. Unsupervised learning methods, like autoencoders, are really valuable where data annotation can be costly or difficult to obtain. As the autoencoder has modeled the underlying dataspace, we can also use it to create new examples of digits or to generate edge cases between different classes.

Given how well the data is separated by our autoencoder, we can again apply our RF classifier to the reduced dimension data. This is the exact form as used for the PCA, but where we pass the latent representation to our classifier. We find our classifier performance has increased to over 60%. However, the dimensionality of our latent space, or the constraints imposed by our bottleneck, are extreme. If we increase the size of the latent space to, for example, dimension 10, we find a much more significant increase in the performance of the classifier to 80%. These results really highlight the benefit of using generative models to pre-process data for downstream data science tasks.

We have learnt that autoencoders are trained by minimizing a reconstruction loss between the input and output of the network. However, sometimes the model can get confused by similarities between classes. This is useful for our model, it means that we can use it to find edge cases or adversarial examples that might perform badly with our classifier. Examples of these edge cases can be seen in some of the examples in Figure 6.8. While the 0 is well reconstructed, we see that 3, 8 and 7 begin to look like other digits. Later, when we meet variational autoencoders, we will see how we can better structure the latent space and how this leads to better synthetic data. However, before this, we return to graph-structured data and investigate how to implement an autoencoder on graphs.

Figure 6.11 Autoencoders are trained by minimising the reconstruction loss. Well-trained autoencoders produce similar results to the input data but can still be confused by edge cases



6.2.4 Building a Graph AutoEncoder for Link Prediction

So far, we have only been discussing classical data science methods for dimension reduction and, especially, how this can be improved by using autoencoders. The reason that we have yet to discuss graph-based learning is because the autoencoder framework is an incredibly adaptive one. Just as the attention mechanisms in Chapter 3 can be used to improve on many different models, autoencoders can be combined with many different models including different types of GNNs. Once the autoencoder structure is understood, the encoder and decoder can be replaced with any type of NN, including different GNNs such as the GCN and GraphSage architectures from Chapter 2.

However, we need to take care when applying autoencoders to graph-based data. When reconstructing our data, we also have to reconstruct our adjacency matrix. We'll next look at implementing a graph-based autoencoder, or GAE, using the amazon dataset from Chapter 4. We'll define a GAE specifically for the task of link prediction which is a very common problem when working with relational data. This allows us to reconstruct the adjacency matrix, and is especially useful where we are dealing with a dataset that has missing data.

Review of the AMZN dataset from Chapter 4

In Chapter 4, we learnt about the Amazon product co-purchaser dataset. This dataset contains information about a range of different items that were purchased, details about who and how they were purchased, and categories for the items, which were the labels in Chapter 4. We learnt about how we can turn this tabular dataset into a graph structure and, by doing so, our learning algorithms were more efficient and more powerful. It is worth mentioning here that we have already relied on PCA without realizing it. PCA is applied to the Amazon co-purchaser dataset to create the features. Each product description was converted into numerical values using the 'bag-of-words' algorithm and PCA is then applied to reduce the (now numerical) description to 100 features. We'll leave it to you to try out applying the autoencoder methods that we've just learnt on the original dataset and see how that impacts results.

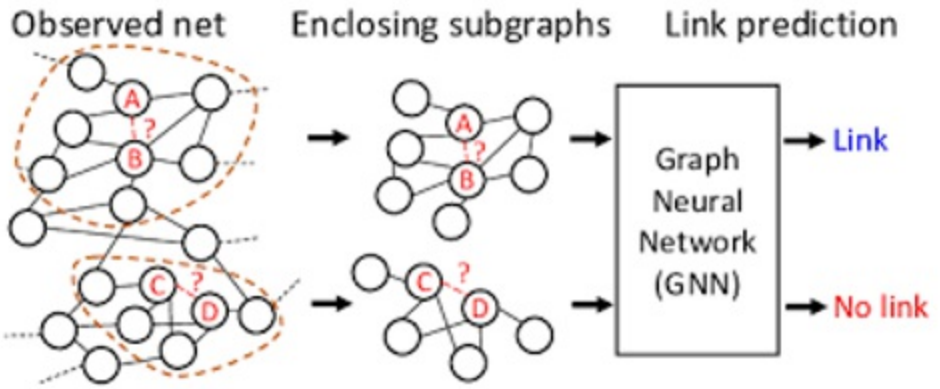
Instead, in this chapter, we're going to revisit the Amazon co-purchaser dataset but with a different aim in mind. First, we are going to take inspiration from the NIST and MNIST datasets and consider only a subset, the category of photography electronics. This is because generative models can be more computationally intensive than discriminative models. This makes sense as they are having to model the entire dataset, rather than just class boundaries as with discriminative models. We are then going to use this dataset to learn link predictions. Essentially, this means learning the *relations between* nodes in our graph which would be Amazon products in our case. This has many important use cases, such as predicting what films or TV shows users would like to watch next, suggesting new connections on social media platforms, or even predicting customers who are more likely to default on credit. In this case, we're going to use it to predict which products in the Amazon Electronics dataset should be connected together.

Understanding link prediction tasks

Link prediction is a common problem in graph-based learning, especially in situations where we have incomplete knowledge of our data. This might be because the graph changes over time, for example where we expect new customers to use an e-commerce service and we want a model that can give the best suggested products to buy at that time. Alternatively, it may be costly to acquire this knowledge. For example, if we want our model to predict

which combinations of drugs lead to specific disease outcomes. Finally, our data may contain incorrect or purposefully hidden details, such as fake accounts on a social media platform. Link prediction allows us to *infer* relations between nodes in our graph. Essentially, this means creating a model that predicts when and how nodes are connected.

Figure 6.12 Link prediction — Schematic explaining how link prediction is performed in practice. Subsections of the input graph (subgraphs) are passed to the GNN with different links missing and the model learns to predict when to recreate a link



For link prediction, our model will take pairs of nodes as input and predict whether these nodes are connected (whether they should be *linked)*. In order to train a model, we will also need ground-truth targets. We generate these by hiding a subset of links within the graph. These hidden links become the missing data that we will learn to infer, which are known as *negative samples*. However, we also need a way to encode the information about pairs of nodes. Both of these parts can be solved simultaneously using graph autoencoders as we set out below.

Defining a Graph AutoEncoder

We are going to use a generative model, the autoencoder, to estimate and predict links in the Amazon computer dataset. In doing so, we're in good company, link prediction was the first problem that GAEs were applied to when first published by Kipf and Welling in 2012. In this seminal paper, they applied the GAE (as well as the variational extension, which we'll be meeting shortly) to three classic benchmarks in graph deep learning, the Cora dataset,

citeseer and PubMed, where we give an overview of some of these different datasets in Chapter 2. However, most graph deep learning libraries now make it very easy to create and begin training graph autoencoders (GAE), as these have become one of the most popular graph-based deep generative models. We'll look at each step in more detail below.

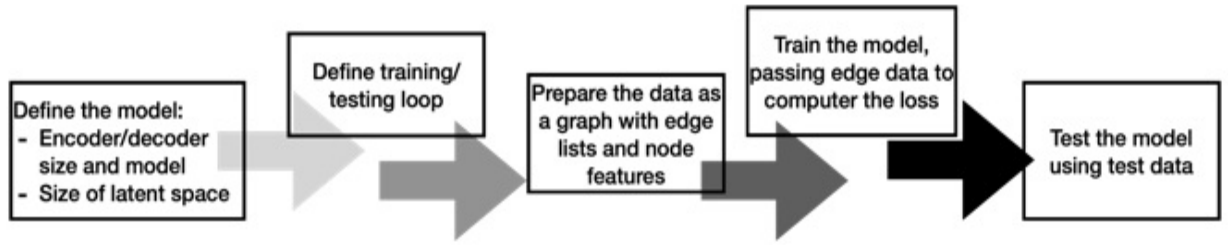
The Graph AutoEncoder (GAE) model is very similar to the typical autoencoder, which we introduced to above. The only difference is that each individual layer of our network is a GNN, such as a GCN or GraphSage network. Code snippets for implementing each step in pyTorch Geometric are given below.

Listing 6.11 Graph Encoder

```
from torch_geometric.nn import GCNConv    #A

class GCNEncoder(torch.nn.Module):    #B
    def __init__(self, input_size, layers, latent_dim):    #B
        super().__init__()
        self.conv0 = GCNConv(input_size, layers[0])    #C
        self.conv1 = GCNConv(layers[0], layers[1])    #C
        self.conv2 = GCNConv(layers[1], latent_dim)    #C

    def forward(self, x, edge_index):    #D
        x = self.conv0(x, edge_index).relu()    #D
        x = self.conv1(x, edge_index).relu()    #D
        return self.conv2(x, edge_index)    #D
```

As we can see above, we first import the relevant models that we will be using to encode our data. Here, we follow the same structure as in Listing 6.5, where we pass the input size, layers, and the size of our latent dimension. However, note that we have to adjust our forward pass to also include the edge data from our graph. This is because we're going to be using our autoencoder to reconstruct the graph from the latent space. For an overview of this, see Figure 6.13. This is also why link prediction is a great application for autoencoders. The autoencoder is learning how to reconstruct the adjacency matrix from a low-dimensional representation of our feature space. This means it is also learning to predict edges when passed new data. However, we have to alter the autoencoder structure to allow it to reconstruct edges. This means we need to change the decoder.

Figure 6.13 Overall steps for training our model for link prediction



The Inner Product Decoder

Below we give a code snippet for the inner product decoder used in graph autoencoders. This is significantly different from the decoder structure that we met earlier in the chapter. This is because we're reconstructing the *adjacency matrix* from the latent representation of our feature data. The graph autoencoder is learning how to rebuild the graph (to infer the edges) given a latent representation of the nodes. We can use the inner product, re-scaled by the sigmoid function, to gain a probability for a node between edges. This allows us to build a decoder that takes samples from our latent space and returns probabilities of whether an edge exists, namely perform edge prediction. This is shown below, in Listing 6.12.

Listing 6.12 Graph Decoder

```
class InnerProductDecoder(torch.nn.Module):    #A
    def __init__(self):    #A
         super().__init__()    #A

    def forward(self, z, edge_index):    #B
     value = (z[edge_index[0]] * z[edge_index[1]]).sum(dim=1)
      return torch.sigmoid(value)
```

The inner product decoder works by taking the latent representation of our data and applying the inner product of this data using the passed edge index of our data. We then apply the sigmoid function to this value which returns a matrix where each value represents the probability that there is an edge between the two nodes. Later, we will apply the binary cross-entropy loss to these values and compare against our true edge index to see whether our model has reconstructed the adjacency matrix correctly. We'll discuss in

more detail why we're using a binary cross-entropy later in the chapter.

Finally, we combine all these together as before, in a graph autoencoder class which contains both encoder layer and decoder layer. Note now we do not initialise the decoder with any input or output sizes as this is just appling the inner product to the output of our encoder with the edge data.

Listing 6.13 Graph AutoEncoder

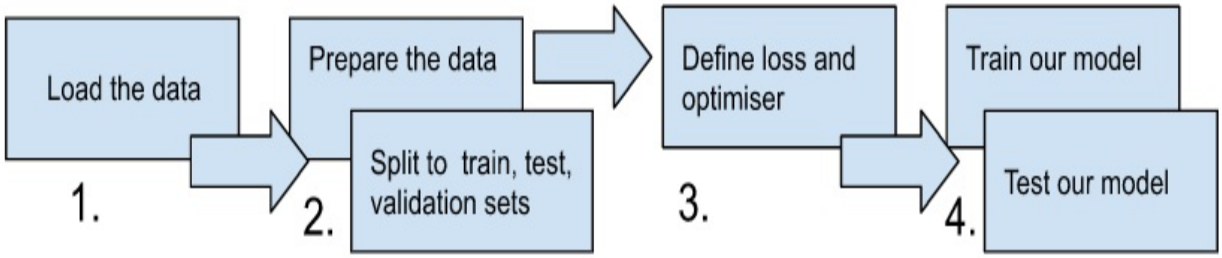
```
class GraphAutoEncoder(torch.nn.Module):
     def __init__(self, input_size, layers, latent_dims):
      super().__init__()
     self.encoder = GCNEncoder(input_size, layers, latent_dims)
      self.decoder = InnerProductDecoder()    #B

     def forward(self, x):
      z = self.encoder(x)
      return self.decoder(z)
```

Above, we can see how the structure of the graph autoencoder is very similar to the autoencoder we built earlier in this book. In Torch Geometric, this structure is made more straightforward by the GAE subclass, which automatically builds both decoder and autoencoder once passed the encoder. We will make use of this functionality when we build a variational autoencoder later in the chapter.

Training a Graph AutoEncoder to perform link prediction

Having built our graph autoencoder, we next proceed to use this to perform edge prediction for the Amazon customer dataset. The overall framework will follow a typical deep learning problem format, where we first load the data, prepare the data and split this data into train, test and validation datasets, define our training parameters and then train and test our model. These steps are shown in Figure 6.14 below.

Figure 6.14 Overall steps for training our model for link prediction



We begin by loading the dataset and preparing it for our learning algorithms. This is largely similar to the steps taken in Chapter 4. Once we load our data, we convert it into sparse tensors. Sparse tensors are useful for edge prediction based tasks as well as many others in geometric learning as tensors often contain many zeros. By converting them to sparse tensors, we store and operate on them much more efficiently. Here, we use the Torch package, Torch Sparse to convert our adjacency matrix into a sparse tensor.

Listing 6.14 Convert to sparse tensor

```
from torch_sparse import SparseTensor #A

edge_index = adj_matrix.nonzero(as_tuple=False).t() #B
edge_weight = adj_matrix[edge_index[0], edge_index[1]] #C

adj = SparseTensor(row=edge_index[0], #D
                   col=edge_index[1], #D
                   sparse_sizes=(num_nodes, num_nodes)) #D
```

We then load the PyG libraries and convert our data into a PyG data object. We then apply transformations to our dataset, where the features and adjacency matrix are loaded as in Chapter 4. First we normalise our features and then we split our dataset into training, testing, and validation sets based on the *edges* or *links* of the graph. This is a vital step when carrying out link prediction to ensure we correctly split our data. In the code, we have used 5% of the data for validation and 10% for test data, noting that our data is undirected. Here, we do not add any negative training samples.

Listing 6.15 Convert to Torch Geometric object

```
data = Data(x=feature_matrix, y=labels, adj_t=adj) #A
transform = T.Compose([
       T.NormalizeFeatures(),#B
```

```
        T.RandomLinkSplit(num_val=0.05, num_test=0.1, is_undirecte
                          add_negative_train_samples=False)]) #B
train_data, val_data, test_data = transform(data)
```

With everything in place, we can now apply GAE to the AMZN computer dataset. First, we define our model, as well as our optimiser and our loss.

Listing 6.16 Define model

```
input_size, latent_dims = num_features, 16 #A
layers = [512, 256] #A
model = GraphAutoEncoder(input_size, layers, latent_dims) #B
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.BCEWithLogitsLoss() #C
```

It is important to note that we have used a binary cross-entropy loss here. This is because we want to calculate the probabilities that each edge is a true edge, where true edges correspond to the ones that we aren't hiding or need to be predicted (i.e. the positive samples). The encoder learns to compress the edge data but doesn't change the number of edges while the decoder learns to both predict edges. In a sense, we're combining both the discriminative and generative steps here. Hence, the binary cross-entropy gives us a probability where there is likely to be an edge between these nodes. It is binary as either an edge should exist (label 1) or shouldn't (label 0). We can compare all those edges that have a binary cross-entropy probability greater than 0.5 to the actual true edges that we have during each epoch of our training loop. This can be seen in the code snippet below.

Listing 6.17 Training function

```
def train(model, criterion, optimizer):

    model.train()

    optimizer.zero_grad()
    z = model.encoder(train_data.x, train_data.edge_index) #A

    neg_edge_index = negative_sampling( #B
    edge_index=train_data.edge_index, num_nodes=train_data.num_no
    num_neg_samples=train_data.edge_label_index.size(1), method='

    edge_label_index = torch.cat( #C
```

```
    [train_data.edge_label_index, neg_edge_index], #C
    dim=-1,) #C

    out = model.decoder(z, edge_label_index).view(-1) #D

    edge_label = torch.cat([ #E
    train_data.edge_label,#E
    train_data.edge_label.new_zeros(neg_edge_index.size(1))#E
    ], dim=0)#E

    loss = criterion(out, edge_label) #F
    loss.backward() #F
    optimizer.step()

    return loss
```

Here, we first encoded our graph into a latent representation. We then perform a round of negative sampling, with new samples drawn for each epoch. Once we have these new negatives samples, we concatenate them with our original edge labels index and pass these to our decoder to get a reconstructed graph. Finally, we concatenate our true edge labels with the 0 labels for our negative edges and we compute the loss between our predicted edges and our true edges. Note, we are not doing batch learning here, instead choosing to train on all data during each epoch.

Our test function is much simpler than our training function as it does not have to perform any negative sampling. Instead, we just use the true and predicted edges and return a ROC-AUC score to measure the accuracy of our model. We discussed ROC-AUC curves in more detail in Chapter 5. Recall that the ROC-AUC curves will range between 0 to 1 and a perfect model, whose predictions are 100% correct, will have an AUC of 1.

Listing 6.18 Test function

```
@torch.no_grad()
def test(data):
    model.eval()
    z = model.encode(data.x, data.edge_index) #A
    out = model.decode(z, data.edge_label_index).view(-1).sigmoid
    loss = roc_auc_score(data.edge_label.cpu().numpy(),#C
                         out.cpu().numpy()) #C
    return loss
```

At each timestep, we will calculate the overall success of a model using all our edge data from our validation data. After training is complete, we then use the test data to calculate the final test accuracy.

Listing 6.19 Training loop

```
best_val_auc = final_test_auc = 0
for epoch in range(1, 201):
    loss = train(model, criterion, optimizer) #A
    val_auc = test(val_data) #B
    if val_auc > best_val_auc:
        best_val_auc = val_auc
test_auc = test(test_data) #C
```

We find that after 200 epochs, we achieve an accuracy of more than 83%. Even better, when we then use our test set to see how well our model is able to predict edges, we get an accuracy of 86%. We can interpret our model performance as being able to suggest a meaningful item to the purchaser 86% of the time, assuming that all future data is the same as our dataset. This is a great result and demonstrates how useful graph neural networks are for systems such as recommender systems. Even better, we can also use our model to better understand how the dataset is structured or apply additional classification and feature engineering tasks, by exploring our newly constructed latent space. However, before we consider this, we're first going to learn about one of the most common extensions to the autoencoder model, the variational autoencoder.

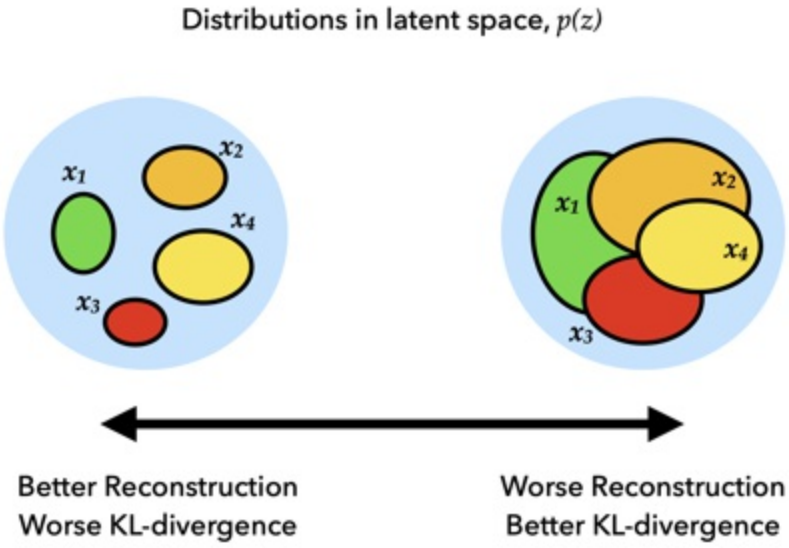
6.3 Variational AutoEncoders

So far, we've discussed how generative models differ from discriminative models and worked through building a model with bottleneck-style architecture. We applied this to classifying hand-written digits and the link prediction problem of how to identify which products are bought together using the Amazon co-purchaser dataset (which we first met in Chapter 4). All of these examples used the autoencoder architecture. As discussed in the preceding sections, autoencoders map data onto discrete points in the latent space. To sample outside of the training dataset, and generate new synthetic data, we can interpolate between these discrete points. This is exactly the

process that we described in Figure 6.1, where we generated unseen combinations of data such as a flying boat. However, because autoencoders are discrete, this can lead to sharp discontinuities in the latent space and this can affect performance for data generation resulting in synthetic data that does not reproduce the original dataset as well.

Returning to our paper-loving business owner, suppose we have a few examples of text written using a different alphabet. These are initially discounted from our paper-text conversion pipeline but the business owner decides he would also like these converted. There are too few instances of these letters to train a classifier outright so we instead decide to generate new instances. However, when we do so, we find that the synthetic images are even more illegible than our ground-truth examples. In order to improve our generative process, we need to ensure that our latent space is well-structured, or *regular*.

Figure 6.15 Regular spaces are continuous and compact, but data regions may become less separated. Alternatively, high reconstruction loss typically means data is well separated but the latent space might be less covered leading to worse generative samples,



Regular means that the space fulfills two properties: continuity and compactness. Continuity means that nearby points in the latent space are decoded into approximately similar things, while compactness means that any point in the latent space should lead to a meaningful decoded

representation. These terms, approximately similar and meaningful, have precise definitions which you can read more about [here]. However, for this chapter, all you need to know is that these properties make it easier to sample from the latent space, resulting in cleaner generated samples and potentially higher model accuracy.

When we regularise a latent space, we use variational methods which model the entire data space in terms of probability distributions (or densities). As we shall see, the main benefit from using variational methods is that the latent space is well structured. However, variational methods do not necessarily guarantee higher performance so it's often important to test both autoencoder and the variational counterpart when using these types of models.

In the next few sections we'll go into more detail on what it means to model a dataspace as a probability density and how we can transform our autoencoder (AE) into a variational autoencoder (VAE) with just a few lines. We'll give a few high-level descriptions of key probabilistic machine learning concepts such as the *Kullback-Leibler divergence* and *the reparameterization trick* but we refer the reader to [*Probabilistic Deep Learning - Manning Press*] for a more thorough treatment. Finally, we'll wrap this section up by demonstrating how we can apply VAEs to graph-based data, introducing the Variational Graph Auto-Encoder (VGAE) architecture and applying it to the same AMZN dataset as before.

6.3.1 Applying a VAE to the MNIST data

Variational inference aims to approximate the conditional probability of a *latent* variable given some observables. We can understand this in terms of our encoder-decoder structure where we are trying to learn the probability of a variable in the latent space given the input data. Returning to our digits examples above, we use variational inference when we approximate the probability of a point in our latent space given the image of the digit. Predicting probabilities, rather than the points directly, implies that we are sampling from probability *densities* parameterized by the free variational parameters. When we train our models, we are optimizing these parameters in order to best match our sampling density to the target density. In the case above, we will be tuning our variational parameters such that the trained

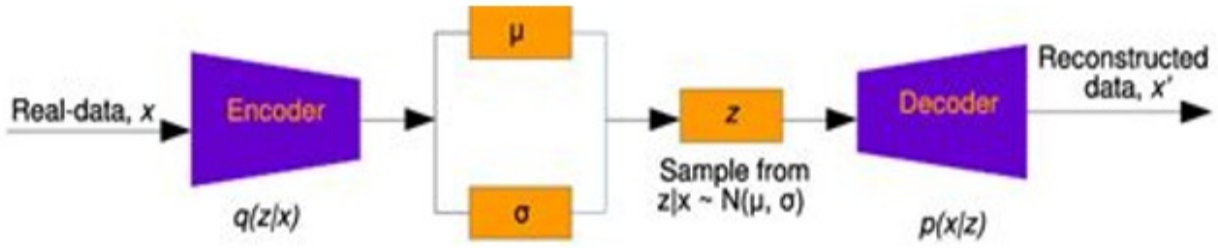
probability density (contained in the latent space) returns correct images when we sample from it. When we combine this with the autoencoder structure, we create a variational autoencoder (or VAE).

For autoencoders, as we will show, this requires a relatively small change in the architecture. Previously, we used a bottleneck structure where neural network layers took our input data and collapsed it onto a low-dimensional space. When we applied this to the digit dataset, we compressed the digits into a two or ten-dimensional space to help us classify them. In the case where the output from the encoder was two channels (representing both dimensions in the latent space), for VAEs the output dimension would be four (a mean and variance for each dimension of the latent space).

Structure of a variational autoencoder

We're now ready to introduce the VAE architecture. As described, this architecture is very similar to the AE model we described in Section 6.2. The main difference is that the output of a variational encoder is generated by sampling from a probability density. We can characterise a density in terms of its mean and variance. Hence, the output of the encoder will now be the mean and variance for each dimension of our previous space. The decoder then takes this sampled latent representation and decodes it to appear like the input data. This can be seen in Figure 6.16.

Figure 6.16 Structure of a variational autoencoder



To make this clearer, we're going to continue with the example of classifying digits (using the MNIST dataset) and put together the different parts of our network. The main difference with this network and the previous autoencoder network is that our latent space now consists of probability densities that we

sample from to generate latent representations. In order to create this, we make our encoder return the mean and (log) variance that parametrises the probability density. Note, the mean and (log) variance can be high dimension leading to very expressive densities. This means that we have to change the dimensionality of the output channel for our encoder and the input channel for our decoder. This is shown in the PyTorch code block below:

Listing 6.20 (Variational Encoder Layer)

```
class VariationalEncoder(nn.Module):
    def __init__(self, input_size, layers, latent_dims):
      super().__init__()
      self.layer0 = nn.Linear(input_size, layers[0])
      self.layer1 = nn.Linear(layers[0], layers[1])
      self.layer2 = nn.Linear(layers[1], latent_dims*2) #A

    def forward(self, X):
      X = torch.flatten(X, start_dim=1)
      X = F.relu(self.layer0(X))
      X = F.relu(self.layer1(X))
      X = self.layer2(X).view(-1, 2, latent_dims) #B

      mu = X[:, 0, :] #C
      logvar = X[:, 1, :] #D
      return mu, logvar
```

Using the Reparameterization Trick to train VAEs

As we can see, the variational encoder model is very similar to our previous encoder, given in Listing 1.5. However, our latent presentations, the variables in the latent space, are being generated by sampling from a distribution. This introduces a stochastic component into our model which causes difficulties when we backpropagate during training. In order to remove these stochastic effects, we use the reparameterization trick. This allows the gradient from the stochastic part of the network to also be back-propagated.

To do so, we separate out the stochasticity into a new variable, which we call epsilon. This means that mu and the log variance remain as learnable parameters during training. The epsilon term is sampled from a normal distribution with mean of zero and variance of one and the latent

representation is found by multiplying the variance by epsilon and adding the mu. We can see this explicitly in the code snippet below:

Listing 6.21 Variational AutoEncoder Layer

```
class VariationalAutoEncoder(nn.Module):
    def __init__(self, input_size, layers, latent_dims):
        super().__init__()
        self.encoder = VariationalEncoder(input_size, layers, lat
        self.decoder = Decoder(latent_dims, layers, input_size)

    def reparametrize(self, mu, logvar):
      if self.training: #A
              z = torch.randn_like(sigma)  #B
              sigma = torch.exp(0.5*logvar) #C
              return z.mul(sigma).add_(mu) #D
      else:
                return mu #D

       def forward(self, x):
          mu, logvar = self.encoder(x)
          z = self.reparametrize(mu, logvar)
          return self.decoder(z), mu, logvar
```

Most of the code snippet above is the same as Listing 1.6. We use a variational encoder, with increased output size, and we use the reparametrisation trick to effectively train our network. The decoder learns to decode the latent representation but we also return the mean and log variance of our distribution. This is because we need both to compute our loss which has an additional component compared to the autoencoder.

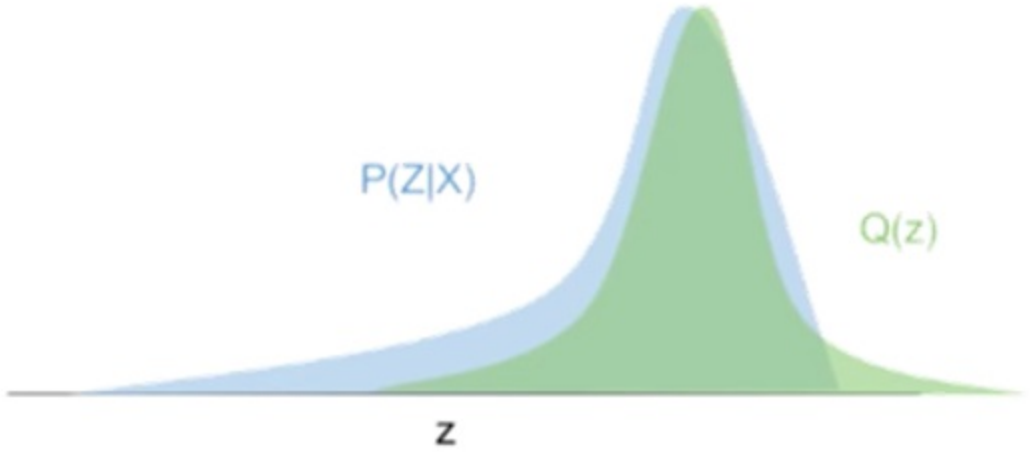
In order to train a VAE, and perform variational inference, we need a way to say when our target density is different to the true conditional density. There are many different ways to compare between probability densities but one of the most popular measures is known as the Kullback-Leibler, or KL, divergence.

Regularizing the latent space

Put plainly, the KL divergence tells us how much worse we would be doing if we use the wrong probability density. We can understand this in the context

of coin flips. Suppose we have two coins and we want to guess how well one coin (which we know is fair) matches the other coin (which we don't know is fair). We're trying to use the coin with known probability to predict the probability for the coin we don't. If it's a good predictor (the unknown coin actually is fair), then the KL-divergence will be zero. The probability densities of both coins are the same. But, if we find that the coin is a bad predictor then the KL divergence will be large. This is because the two probability densities will be far from each other. In Figure 6.17 we can see this explicitly. We are trying to model the unknown probability density Q(z) using the conditional probability density P(Z|X). As the densities overlap, the KL divergence here will be low.

Figure 6.17 Kullback-Leibler (KL) divergence which calculates the degree by which two probability densities are distinct. High KL divergence means that they are well separated whereas low KL divergence means that they are not.



Practically, we convert our AE to a VAE by introducing the KL divergence in the loss. Our intention here is to both minimise the discrepancy between our encoder and decoder as in the AE loss but also minimise the difference between the probability distribution given by our encoder and the `true' distribution that was used to generate our data. This is done by adding the KL divergence to the loss. For many standard variational autoencoders, this is given by, where (p||q) denotes the divergence of probability p with respect to probability q:

KL(p||q) = -0.5(1 + logvar - mu^2 - sum(exp(logvar))

In our previous example, this results in a new loss function which we add to our training and test functions. Here, we repeat just the evaluation function but the train function is very similar. We are only the output from our model (to include mean and log variance) and adding the KL divergence to our loss function.

Listing 6.22 Test function

```
@torch.no_grad()
def evaluate(model, iterator, criterion, optimizer):
    model.eval()

    epoch_loss = 0

    for x,label in iterator:

        xbar, mu, logvar = model(x) #A

        xbar = xbar.reshape((-1, 1, 28, 28)) #B

        kl_loss = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logva
        loss = criterion(xbar, x) + kl_loss #D

        #update the loss
        epoch_loss += loss.item()

        print(f"test loss: {epoch_loss /len(iterator):3.7}")
```

We now have our fully completed variational autoencoder. We'll take our new model and apply it to the MNIST dataset. We'll also use this model structure to build a new synthetic data which can be used to test on edge cases.

Variational autoencoders are better generative models

With these three simple changes, we now have a *variational* autoencoder. Most of the implementation from our previous example on the MNIST dataset remains exactly the same. The only differences has been that we've increased the output dimensionality of our encoder, we've used the reparameterization trick to allow us to more effectively learn the mean and log variance, and we change the loss to make sure our latent space is well

structured.

We now retrain the model to produce a two dimensional latent space. We can see output for how the variational autoencoder has structured the latent space, and compare these against the representation from the PCA and autoencoder, as shown in Figure 6.14. When we use these two dimensions to train a classifier on the digits we get an accuracy of 65%. Even better, if we increase the dimensionality of the latent space to size ten, then the accuracy again increases to 80%. This is similar to what we achieved with the autoencoder, where we have the different accuracy scores given in Table 6.1.

Table 6.1 Performance of different methods from this chapter.

Method	PCA	PCA	AE	AE	VAE	VAE
Number of dimensions	2	10	2	10	2	10
Accuracy	50%	55%	60%	80%	65%	80%

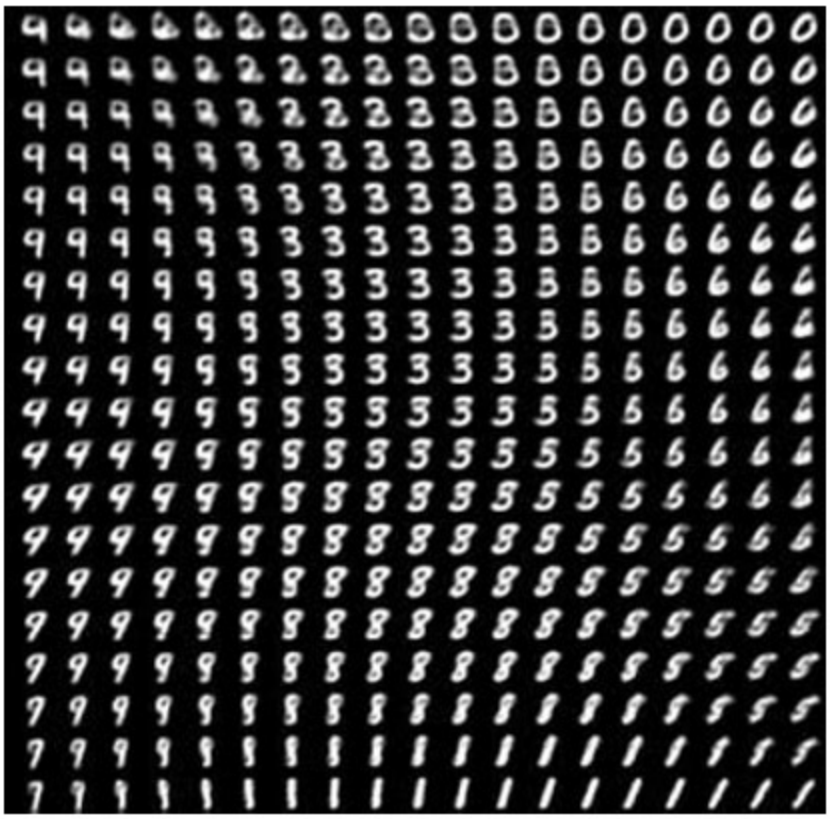
The additional benefit of training a VAE, rather than just an autoencoder, is that the generative abilities can improve. In Figure 6.18 we show a comparison between images generated with our AE and with our VAE. We see that the generated images become sharper and less similar to other digits. Hence, it is always good to consider variational autoencoders where you need to generate new data.

Figure 6.18 Comparison between AE and VAE

We have learnt how variational autoencoders give us extra control over how the latent space, the low-dimensional space that our model uses to represent the data. Specifically, introducing regularising terms through the KL divergence loss term, allows us to improve how we separate out different classes and can be helpful for downstream data science tasks such as clustering or classification tasks. Moreover, a well structured latent space means gives us the possibility for making much higher fidelity synthetic datapoints.

This can be seen in Figure 6.18. In order to generate each square in the image, we interpolate between classes in our dataspace. To do so, we take the latent representation for specific digits, such as a 4, 1 and then sample at each individual value between the encoded representation. The result allows us to see how well the model has learnt the dataspace and can even be used to generate edge cases that are quite similar to two or three classes. We've circled a couple of these examples in Figure 6.19 to make this clearer. Once we have these edge cases, we can check to see how well our classifier performs against them and, if poorly, we can try to improve performance by training on many similar adversarial cases.

Figure 6.19 Interpolating in the Latent Space - need to redo this



6.3.2 Variational Graph AutoEncoders

As we have seen, a variational autoencoder is a relatively straightforward extension of the autoencoder which converts the modelled dataspace from a point-wise discrete approximation to one made of probability densities. This requires two changes to our learning process, we have to adapt our model architecture and also change our loss to include an additional term for regularsing the latent space. This also holds true for the Variational Graph AutoEncoder (VGAE).

Building a Variational Graph AutoEncoder

Below we give a code snippet for the variational autoencoder. The similarities between Listing 6.20 and the variational encoder layer, in Listing 6.23, should be clear: We have doubled the dimensionality of our latent space and are returning from our encoder the mean of the convolutional layer and the log variance. These are returned at the end of our forward pass.

Listing 6.23 Variational GCNEncoder

```
class VariationalGCNEncoder(torch.nn.Module): #A
    def __init__(self, input_size, layers, latent_dims):
      super().__init__()
      self.layer0 = GCNConv(input_size, layers[0])
      self.layer1 = GCNConv(layers[0], layers[1])
      self.layer2 = GCNConv(layers[1], 2*latent_dims)
      self.conv_mu = GCNConv(2 * latent_dims, latent_dims) #A
      self.conv_logvar = GCNConv(2 * latent_dims, latent_dims)#A

    def forward(self, x, edge_index):
      x = self.layer0(x, edge_index).relu()
      x = self.layer1(x, edge_index).relu()
      x = self.layer2(x, edge_index).relu()
      return self.conv_mu(x, edge_index),
 self.conv_logvar(x, edge_index) #B
```

When we discussed the graph autoencoder, we learnt that the decoder structure differs from the typical autoencoder, in that it takes the inner product to return the adjacency matrix, or edge list. Previously we explicitly implemented the inner dot product. However, in PyTorch Geometric, this functionality is inbuilt. To build a variational autoencoder structure, we can call the VGAE function. This is shown below.

Listing 6.24 Variational Graph AutoEncoder (VAE)

```
from torch_geometric.nn import VGAE #A

model = VGAE(VariationalGCNEncoder(input_size, layers, latent_dim
```

This functionality makes it much simpler to build a VGAE, given a variational graph encoder. It can also be used to build a graph autoencoder in exactly the same way as above. Note, the VGAE function in PyTorch Geometric also takes care of the reparameterization trick. Now that we have our VGAE model, the next thing we need to do is amend the training and testing functions to include the KL divergence loss.

Listing 6.25 Training function

```
def train(model, criterion, optimizer):
```

```
    model.train()
    optimizer.zero_grad()
    z = model.encoder(train_data.x, train_data.edge_index)

    neg_edge_index = negative_sampling(
    edge_index=train_data.edge_index, num_nodes=train_data.num_no
    num_neg_samples=train_data.edge_label_index.size(1), method='

    edge_label_index = torch.cat(
    [train_data.edge_label_index, neg_edge_index],
    dim=-1,)
    out = model.decoder(z, edge_label_index).view(-1)

    edge_label = torch.cat([
    train_data.edge_label,
    train_data.edge_label.new_zeros(neg_edge_index.size(1))
    ], dim=0)

    loss = criterion(out, edge_label)  #A
+ (1 / train_data.num_nodes) * model.kl_loss() #A

    loss.backward()
    optimizer.step()

    return loss
```

This is the same training loop that we used in Listing 1.17 to train our graph autoencoder. The only difference is that we include an additional term to our loss which minimise the KL divergence. Otherwise the training remains unchanged. Note that thanks to the added PyTorch Geometric functionality, these changes are considerably less involved than when we made the changes in PyTorch earlier. However, going through each of those extra steps should have given you more intuition for what the changes are doing.

We can now apply our VGAE to the AMZN co-purchaser computer dataset and use this to perform edge prediction. When we do so, we find that the overall test accuracy to be …. This is slightly lower than our accuracy for GAE. This is an important point to note, that VGAEs will not necessarily give higher accuracy. As a result, you should always try a GAE as well as a VGAE and run care model validation when using this architecture.

When to use a Variational Graph Auto-Encoder

Given that the accuracy for the VGAE was slightly lower than the equivalent accuracy for the GAE, it is important to realise the limitations of both methods. In general, GAEs and VGAEs are great models to use when you want to build a generative model or where you want to use one aspect of your data to learn another aspect. For example, we might want to make a graph-based model for pose prediction. We can use both GAE and VGAE architectures to predict future poses based on video footage. We'll see a similar example to this in later chapters. When we do so, we're using the GAE/VGAE to learn a graph of the body, conditioned on what the future positions of where each body part will be. However, if we are specifically interested in generating new data, such as new chemical graphs for drug discovery, variational GAEs are often better as the latent space is more structured. As always, your choice of architecture depends on the problem at hand.

Figure 6.20 The latent space of the AMZN co-purchaser dataset

In this chapter we have learnt about generative models, two specific examples of generative models, the autoencoder and variational autoencoder models, and how to implement these models to work with graph-structured data. In order to better understand how to use this model class, the graph autoencoder

and variational graph autoencoder, we applied our models to an edge prediction task. However, this is only one step in applying a generative model. In many instances where we require a generative model, we use successive layers of autoencoders to further reduce the dimensionality of our system and increase our reconstruction power. In the context of drug discovery and chemical science, graph autoencoders allow us to reconstruct the adjacency matrix (as we did here) as well as reconstruct types of molecules and even the number of molecules. They are being frequently used in many sciences and industries. Now you have tools to try them out too.

6.4 Summary

- Discriminative models learn to separate out data classes while generative models learn to model the entire dataspace.
- Generative models are often used to perform dimension reduction. Principal Component Analysis (PCA) is a form of linear dimension reduction.
- Autoencoders contain two key components, the encoder and the decoder, both represented by neural networks. They learn how to take data and encode (compress) it into a low dimensional representation as well as decode (uncompress) it again. For autoencoders, the low dimensional representation is known as the latent space.
- Variational autoencoders extend autoencoders to have a regularizing term in the loss. This regularizing term is typically the Kulbeck-Liebler (KL) divergence which measures the distance between two distributions. The latent space of variational autoencoders is made of densities.
- Autoencoders and variational autoencoders can also be applied to graphs. These are, respectively, graph autoencoders (GAE) and variational graph autoencoders (VGAE). They are very similar to typical AEs and VAEs but the decoder element is typically the dot product applied to the edge list.
- GAE and VGAEs are useful for edge prediction tasks. They can help us predict where there might be hidden edges in our graph.

6.5 References

Amazon Product Dataset

McAuley, Julian, Rahul Pandey, and Jure Leskovec. "Inferring networks of substitutable and complementary products." *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015.

Variational Autoencoders

Kingma, D. P., & Welling, M. (2013). Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114.

Variational Graph Autoencoders

Kipf, T. N., & Welling, M. (2016). Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*.

Appendix A. Discovering Graphs

This appendix covers

- The elements of graph and network theory relevant to GNNs
- Understanding common graph representations, data models, and data structures
- Introducing the graph ecosystem, including databases, graph processing systems, and libraries
- Understanding graph algorithms and their relevance to graph neural networks
- Guidelines on reading graph academic literature

In this appendix, we delve into the theory and implementations of graphs that are most pertinent to using the GNNs covered in the rest of the book. For theory, we establish basic definitions, concepts and nomenclature. We then survey how the theory is realized in real systems. This foundation is not only necessary to follow the advanced materials in subsequent chapters, but in building the insights that make architecting custom systems and troubleshooting errors easier.

In addition, in a highly evolving field, being able to absorb new academic and technical literature is a critical asset in getting up to speed quickly on the state of the art. This chapter also aims to provide the basic background to be able to pick up the essence of relevant published papers. Advanced readers who are familiar with graphs can skip this.

In this appendix we'll use a running example of a social networking dataset to demonstrate the concepts. This is a dataset of over 1,900 professionals and their industry relationships. Figure A.1 visualizes this graph (generated using Gephi). As we progress in chapters 2 and 3, we'll continue with this example, and learn how to describe this network and its elements using the language of graphs; explore this data using tools in the graph ecosystem; visualize this graph in different ways; and even how to generate it from raw data.

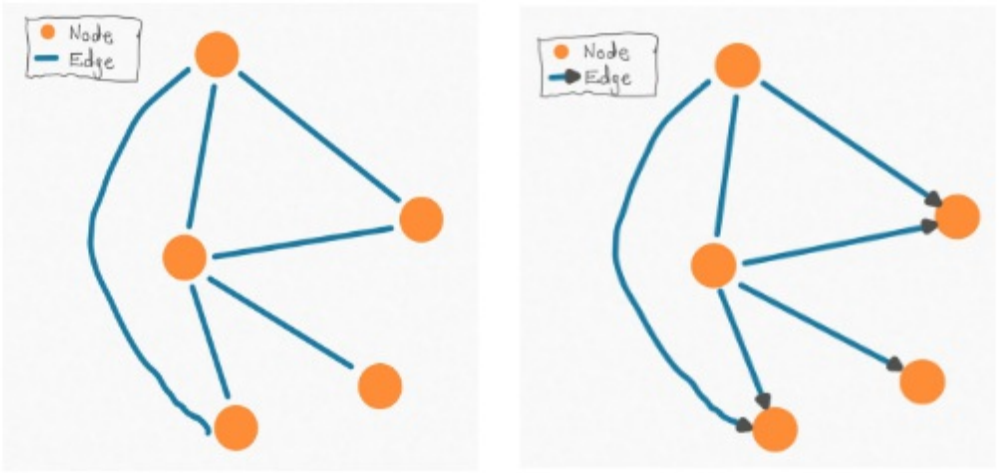
Figure A.1 A stylized visualization of the example social network, consisting of industry professionals and their relationships. The nodes (dots) are the professionals, and the edges (lines) denote a relationship between people. In this visualization, created using Graphistry, the left image shows an edge diverge out of the frame(bottom right). The right image is the entire graph, showing the cut off edges and nodes.

A.1 Graph Fundamentals

A graph is a data type which consists of two main elements: **edges** and **nodes**. Edges represent relationships or links and are usually illustrated as lines or arrows. Nodes are endpoints of edges and are usually visualized as points.

Graphs can be classified as **directed**, meaning that the edges have a distinct direction between a node and an edge; such an edge is visualized as an arrow. In such an arrangement, the node of origin is called the **source** node, and the second node is deemed the **destination** node. Or graphs can be **undirected**, where the edge has no direction**;** such an edge is depicted as a line, without an arrow. Edges can also be self-loops; for such an edge the same node is both the source and destination node. Self-loops can be directed or undirected. You can see some of these basic elements in

Figure A.2 Two basic graphs, undirected (left) and directed (right). Circles denote nodes (or vertices) and lines/arrows denote edges.



In our social graph from figure 1, people are represented by nodes, and their relationships are represented by edges. In our subsequent examples, I have chosen to portray our social network as undirected, but we could have represented it as directed. For instance, when two people have a professional relationship, such an association can be seen as symmetric (he knows her; she knows him); this relationship would be undirected. However, we could imagine a reason to make our representation directed. For instance a relationship could be hierarchical: if a person has a higher status or manages another person, such a relationship could be modeled as a directed edge which starts at a higher status individual and ends at the person of lower status. Another instance could be who reached out to whom in a social network. In networks such as Facebook, and LinkedIn, for connections to be made, someone has to initialize them. A directed edge could flow from the person who requests an online connection to the ‘friended’ person.

Let’s start with some definitions, and then see how the concepts work.

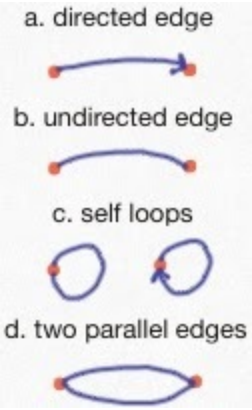
Key Terms

- **Graph -** A data type consisting of nodes and edges.
- **Node -** Also called a **vertex** or **point,** a node is an endpoint in a graph.

They are connected by edges.

· **Edge** - Also called a **link** or **relationship**, an edge connects nodes. They can be directed or undirected.

Figure A.3 Loops and three types of edges



· **Directed Edge** - A directed edge, usually represented by an arrow, denotes a one-way relationship or flow from one node to another.

· **Undirected Edge** - An undirected edge has no direction. In such an edge a relationship or flow can go in either direction.

· **Adjacent** - The property that two nodes are directly connected via an edge. Such nodes are said to be **joined**.

· **Incidence** - The property that a node and an edge are directly connected

· **Self Loop** - An edge that connects a node to itself. Such edges can be directed or undirected.

· **Parallel Edges** - Multiple edges that connect the same two nodes.

· **Weights** - One important attribute of an edge is a weight, a numerical value assigned an edge. Such an attribute can describe the intensity of the connection, or some other real world value, such as length (if a graph modeled cities on a road map).

These concepts give us the tools to create the simplest graphs. With a simple graph created from these concepts, we could derive network properties explained below.

Though real world graphs have more complex structures, for different purposes it is often helpful to use simple graphs to represent them. For example, though our social graph data contains node features (covered in section A.1.2), to create the visualization in figure A.1, I used only node and edge information.

Two types of models

In this book, we'll use the word *model* in two ways, which should be clear given the context. Both uses can be expressed as nouns or verbs.

1 **Machine Learning Model.** Data scientists and machine learning engineers are familiar with the concept of a statistical or machine learning model, which is of course used to discuss GNNs and other models. I'll frequently use *machine learning model*, *statistical model*, or *GNN model* to refer to this usage of *model*.

2 **Graph Data Model.** In the field of networks and graphs, we use *model* to describe the way an abstract or concrete concept can be expressed using a graph structure. This follows the dictionary definition: "a usually miniature representation of something." In this book and in this chapter especially, we talk about how graphs of different types can be used to model or represent real world systems or concepts: road maps, social networks, molecules, etc. For this usage, I'll frequently, but not always, use *graph model*, *graph data model*, or *data model*.

A.1.1 Graph Properties

Below, we discuss some of the more important properties of graphs. Many of the software and databases in the graph ecosystem (described in section A.3) should have the capability to compute some or all of these properties.

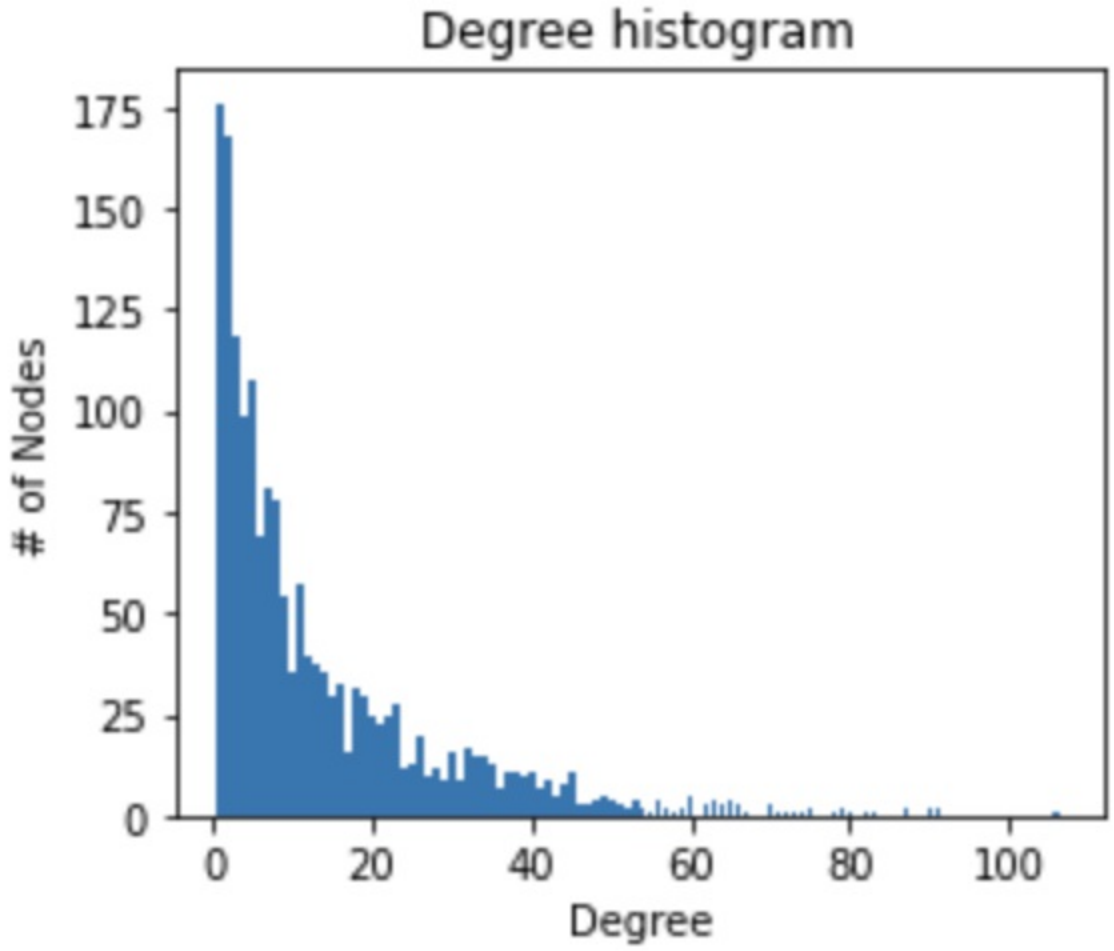
Size/Order. We are often interested in the overall number of nodes and edges in a graph. Formal names for these properties are **size** (the number of edges),

and **order** (the number of nodes.

In our social graph, the number of nodes is 1933, and the number of edges is 12239.

Degree Distribution. A degree distribution is simply the distribution of the degrees of all the nodes in a graph. This is often expressed as a histogram.

Figure A.4 A histogram showing the degree distribution of our social graph



The **degree** of a node is the count of its adjacent nodes in an undirected graph. For directed graphs, there are two types of degrees a node can have: an **in-degree** for edges directed to the node, and an **out-degree**, for edges directed outward from the node. Self loops often are given a count of 2 when calculating degree. If edges are given weights, a **weighted degree** can also

account for these weights.

Related to the concept of a degree is that of a node's neighborhood. For a given node, it's adjacent nodes are also called its **neighbors**. The set of all its neighbors is called its **neighborhood**. The number of vertices in a node's neighborhood is equal to that node's degree.

For our social graph, the figure below expresses the degree distribution as a histogram.

Connectedness. A graph is a set of nodes and edges. In general, however, there is no condition that says for an undirected graph every node can be reached by any other node within the same network. It can happen that within the same graph, sets of nodes are utterly separated from one another; no edge links them.

An undirected graph where any node can reach any other node is called a **connected graph**. It may seem obvious that all graphs must be connected, but this is often not the case. Graphs that have discontinuities (where a node or set of nodes are unlinked to the rest of the graph) are **disconnected graphs**. Another way to think about this is that in a connected graph, there is a path or walk whereby every node can reach every other node in the graph. For a disconnected graph, each disconnected piece is called a **component**. For a directed graph, where it is not always possible to reach any node from any other node, a **strongly connected** graph is one where this is the case.

As an example, the human population of Earth can be considered a disconnected social graph, if we consider individual humans as nodes, and our communication channels as edges. While most of the population can be said to be connected by modern communication channels, there are hermits who chose to live 'off the grid', and isolated hunter-gatherer tribes that reject contact with the rest of the world. In other use cases, it is often the case that there are discontinuities in the network and its data.

Examining our social graph, we see it is disconnected with a large component that contains most of the nodes. Figures A.5 andA.6 show the entire graph, and the large connected component.

If we focus on the large connected component, we find that the size is smaller than the entire graph. The number of nodes is 1698 and the number of edges is 12222.

Figure A.5 Our entire social graph, which is disconnected. We observe a large connected component at the center, surrounded by disconnected nodes and small components consisting of 2-3 nodes. NetworkX was used to generate this figure.

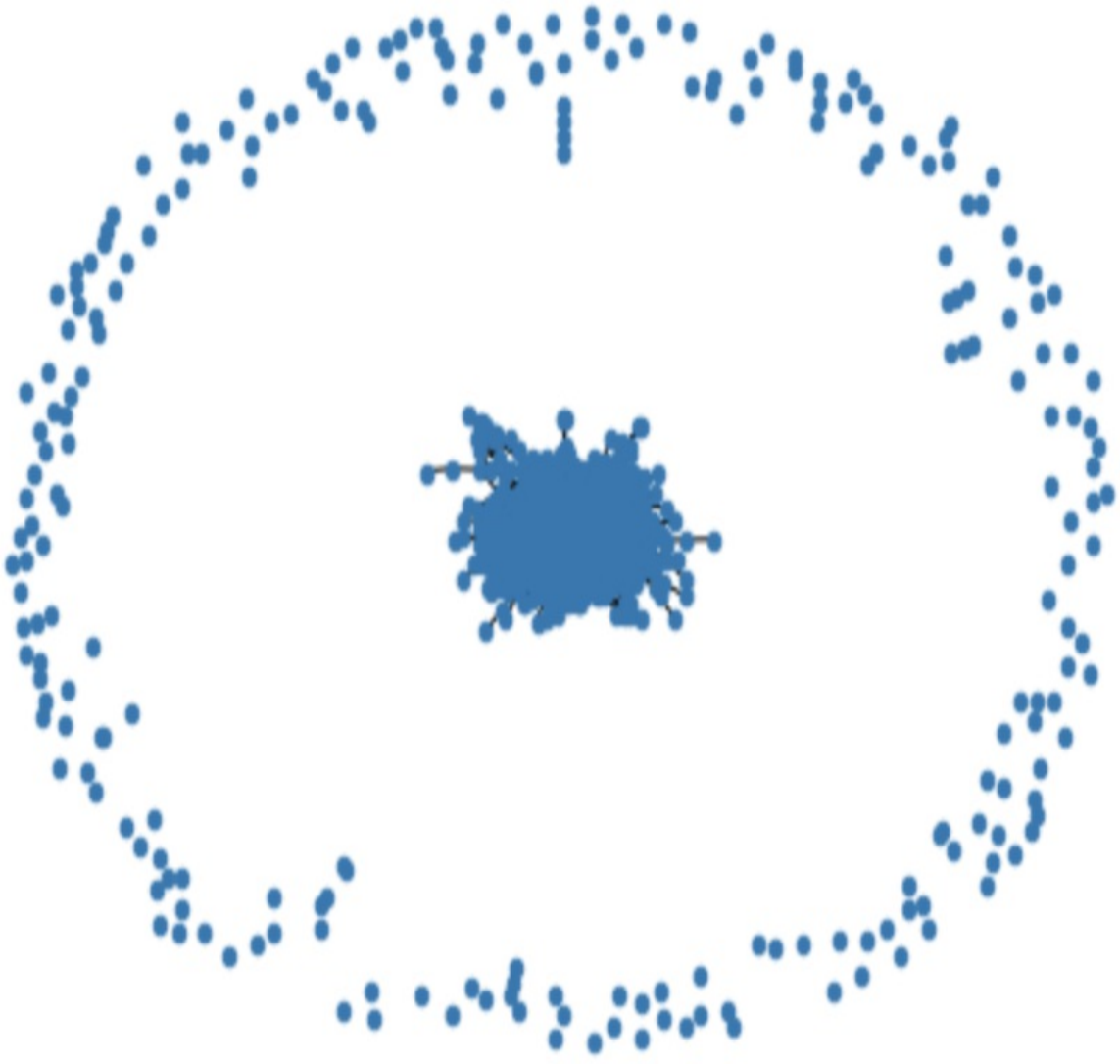
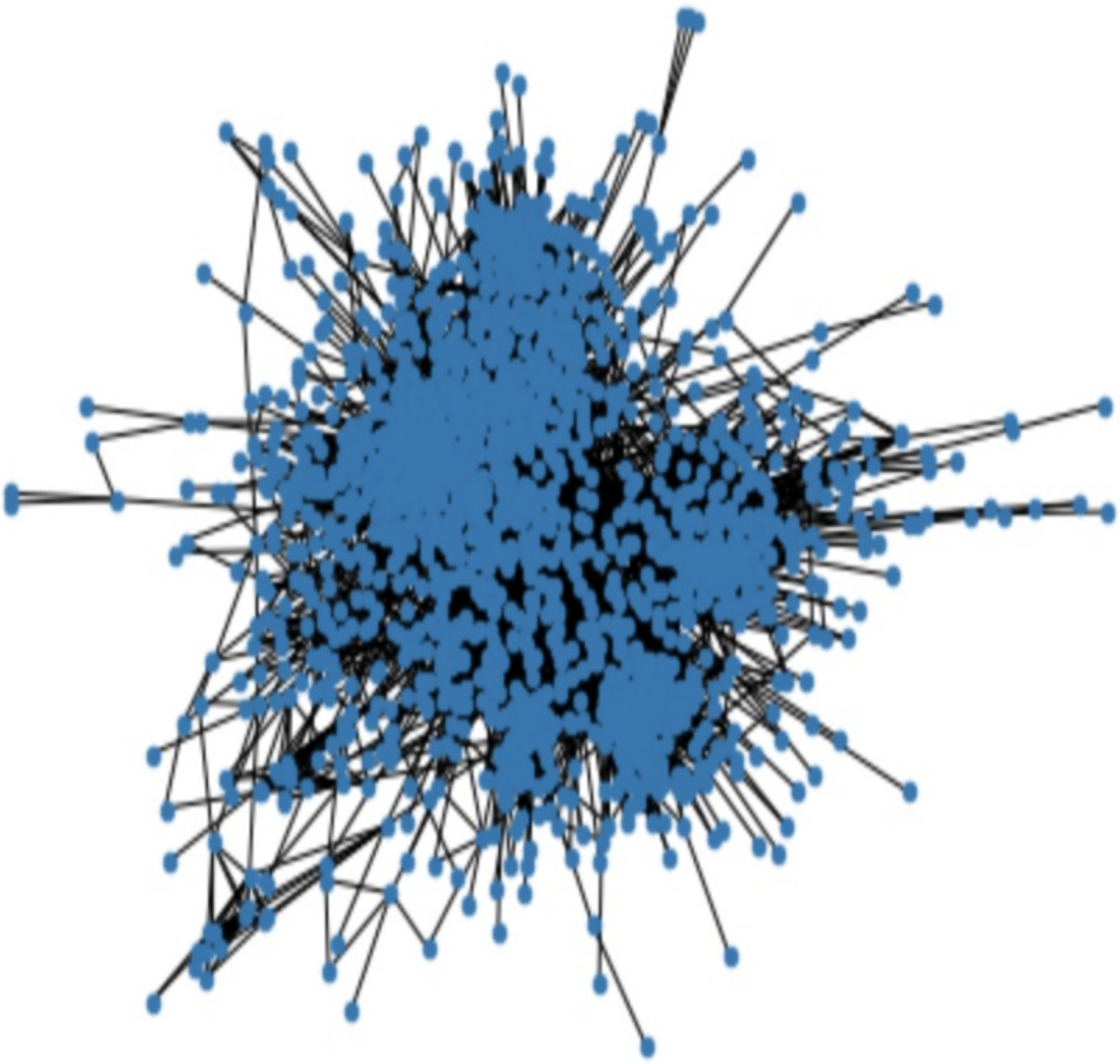


Figure A.6 The connected component of the social graph. NetworkX was used to generate this figure. Compare this to figure 1, which is the same graph visualized using Graphistry. Differences in the parameters used in the algorithms as well as visual features account for the distinctiveness

of the two figures.



Graph Traversals. In a graph, we can imagine traveling from a given node *a* to a second node *b*. Such a trip may require passing only one edge, or it could be a trip where we pass several edges and nodes. Such a trip is called a **traversal**, or a **walk**, among other names.

A walk can be **open** or **closed**. Open walks have an ending node that is different from the starting node. A closed walk starts and ends with the same node.

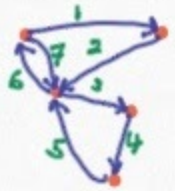
A **path** is a walk where no node is encountered more than once. A **cycle** is a closed path (with the exception of the starting node, which is also the ending node, no node is encountered twice).

A **trail** is a walk where no edge is encountered more than once. A **circuit** is a closed trail.

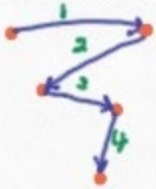
Imagine that for a given pair of nodes, we could find walks and paths between them. Of the paths we could navigate, there will be a shortest one (or maybe more than one path will tie for shortest). The length of this path is called the **distance** or **shortest path length**.

Figure A.7 Five types of walks

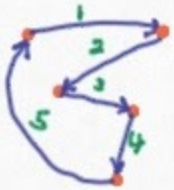
a. walk - traversal along any set of nodes and edges



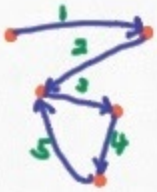
b. path - traversal with no repeated nodes



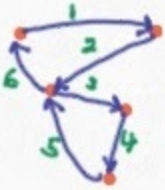
c. cycle - a closed path



d. trail - traversal with no repeated edges



e. circuit - a closed trail



If we zoom out and examine the entire graph and its node pairs, we could list out all of the shortest path lengths. One of these distances will be the longest (or more than one may tie for longest). The largest distance is the **diameter** of the graph. The diameter is often used to characterize and compare graphs.

If we take our list of distances and average them, we'll generate the **average path length** of the graph. Average path length is another important descriptive measure for networks. Both average path length and diameter

give an indication of the density of the network; higher values for these metrics imply more connections, which in turn allow a greater variety of paths, both longer and shorter.

For our social graph, the diameter of our largest component is 10. Diameter is undefined for the entire graph, which is unconnected.

Subgraphs. Consider a graph of nodes and edges. A subgraph is a subset of these nodes and edges. Subgraphs are of importance when these 'neighborhoods' in the graph have properties that are distinct from other locations in the graph. Subgraphs occur in connected and disconnected graphs. A component of a disconnected graph is a subgraph.

Clustering Coefficient. A node may have a high degree, but how well connected is its neighborhood? We can imagine an apartment building where everyone knows the landlord, but no one knows their neighbors (what a sad place!). The landlord would have a clustering coefficient of zero. At the other extreme, we could have an apartment where the landlord knows all the tenants, and every tenant knows every other tenant. Then, the landlord would have a clustering coefficient of 1 (such a situation, where all the nodes in a network are connected to every other node is called a **complete** graph). Of course, there will be intermediate cases where only some of the tenants know one another, these situations will have coefficients between 0 and 1.

The dimension of a graph

In machine learning and engineering in general, *dimension* is used in several ways. This term can be confusing as a result.

Even within the topic of graphs, the term is used in a few ways in articles and academic literature. However, the term is often not explicitly defined or clarified. Thus, below, we attempt to deconstruct the meaning of this term.

1 Size/Shape of Datasets: In this case, the term dimension refers to the number of features in a dataset. Low dimensional datasets are implied to be small enough to visualize (i.e., 2 or 3 features), or small enough to be computationally viable.

2 Mathematical Definitions: In math, the dimension of a graph has more strict definitions. In linear algebra, graphs can be represented in vector spaces, and the dimension is an attribute of these vector spaces [ref].

3 Geometric definition: There is also a geometric definition of a graph's dimension. This definition relates a graph's dimension to the least number of Euclidean dimensions that will allow a graph's edges to be of unit size 1 [ref].

A.1.2 Characteristics of Nodes and Vertices

In the most basic type of graph, we have a collection of nodes and edges, without parallel edges or self loops. For this basic graph, we have a geometric structure only.

While even this basic graph structure is useful, for real world problems and use cases, often more complexity is desired to properly model a situation. To this we can:

1. Reduce the geometric restrictions we established above. Explicitly, these restrictions are:
    - Each edge is incident to two nodes, one on each end of the edge
    - Between two nodes, only one edge can exist
    - No self loops
      With these restrictions relaxed, we are able to more accurately model more situations at the cost of more analytic complexity.
2. Add properties to our graph elements (nodes, edges, the graph itself). A property is descriptive data tied to a specific element. Depending on the context, terms like labels, attributes, decorators are used in place of property.

In this section and the next, we'll discuss the characteristics and variants of nodes, edges and entire graphs.

A.1.3 Node Properties

Names, IDs and Unique Identifiers. A name or an ID is a unique identifier.

Many graph systems will either assign an identifier such as an index to a node, or allow the user to specify an ID. In our social graph, each node has an unique alphanumeric ID.

Labels. Within a graph, nodes may fall within certain classes or groups. For example, a graph modeling a social network may group people by their country of residence (‘USA’, ‘PRC’, ‘Nigeria’), or their level of activity within the network (‘frequent user’, ‘occasional user’). In this way, in contrast to unique identifiers explained above, we’d expect several nodes to share the same label.

Properties/Attributes/Features. Properties that aren’t IDs or labels are usually called properties, attributes or features. While such properties don’t have to be unique to a node, they don’t describe a node class, either. Properties can be structure-based, or based on non-structural qualities.

Structural vs Non-Structural Properties

Structural/Topological Properties. Intrinsic characteristics of a node are related to the node's topological properties and the geometrical structure of the graph in proximity to the node. Two examples are:

- a node’s degree, which we learned above was the number of incident edges it has.
- A node’s **centrality**, which is a measure that indicates how important a node is relative to the nodes in its neighborhood.

By employing graph analytical methods (described in Section 2.4) characteristics of nodes, relative to their local environment, can be gleaned. These can be incorporated into certain GNN problems as features. Node embeddings such as those generated by transductive methods (chapter 3) are another example of a property based on the graph’s local structure.

Non-Structural Properties. These are often based upon real-world attributes. Taking the example of our social graph, we have two categorical properties: a person’s job category (e.g., scientist, marketer, administrator), and the type of company they work for (e.g., Medical, Transportation, Consulting). The examples above are categorical attributes. It is possible to have numerical

attributes, such as *years of experience*, or *average number of direct reports* in all current and past roles.

A.1.4 Edge Properties

Properties for edges mirror those for edges. The most often used and important edge property is that of the edge weight, described earlier.

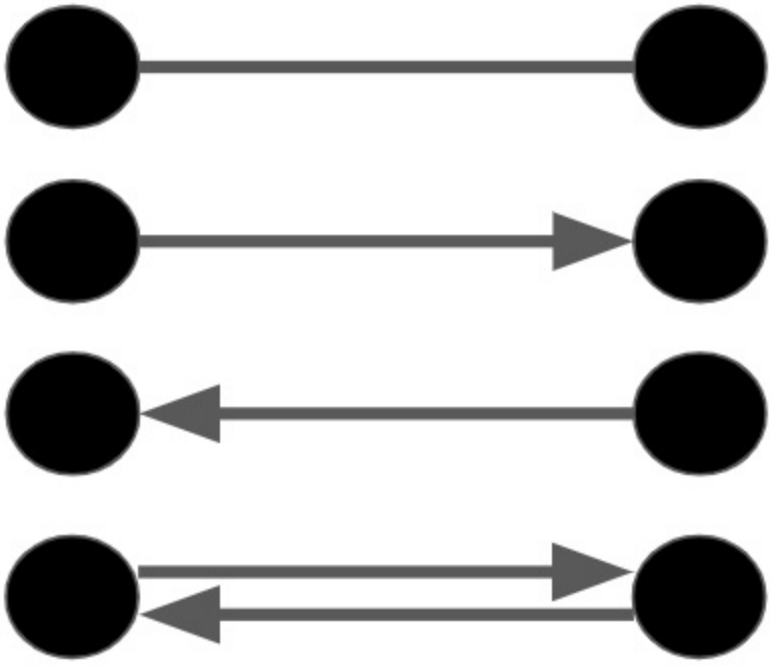
A.1.5 Edge Variations

Unlike nodes, there are a few geometric variants of edges that can be used to make a graph model more descriptive.

Parallel Edges. Meaning more than one edge between two nodes u and v.

Directionality. Edges can have no direction or one direction. Since nodes u and v can have parallel edges connecting them, it is possible to have two edges with opposite directionality, or multiple edges with some combination of directions or undirectionality.

Figure A.8 From top to bottom, between two nodes, an example of an undirected edge, a directed edge from left to right, a directed edge from right to left, and two directed edges traversing both directions (bi-directionality).



There is also a concept of **bi-directionality**, the case where between two nodes, both directions are represented in the respective edges. In practice, this term is used in a few ways:

- To describe non-directed edges, or simple edges.
- To describe two edges that have opposite directions (shown above)
- To describe an edge that has a direction at each end. This usage exists in the literature is seldom encountered in practical systems at this writing.

Self Loops. Discussed above, a self-loop, or loop, is the case where both ends of an edge connect to the same node. Where would one encounter a self-loop in the real world? For our social graph, let's keep all the nodes, and consider a case where an edge would be an email sent from one professional to another. Sometimes people send emails to themselves (for reminders). For such a scenario, an email to oneself could be modeled as a self-loop.

A.1.6 Categories of Graphs

Different categories of graphs depend on the node and edge characteristics described above.

Simple Graph. The formal definition of a **simple graph** is a graph whose edges can't be parallel edges or self-loops. Simple graphs can be connected or disconnected. Also a simple graph can be directed.

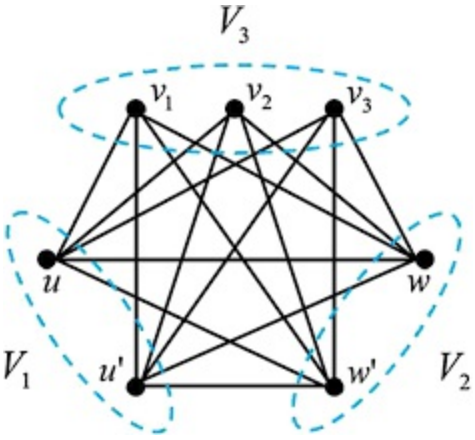
Weighted Graph. A graph that uses weights is called a **weighted graph**. Our social graph has no weights; another way to express having no weights is to set all weights to 1 or 0.

Multigraphs. A **multigraph** is a graph that is permitted to have multiple edges between any two nodes, and multiple self-loops for any one node. A simple graph could be a special case of a multigraph, if we are working within a problem where we could add more edges and self loops to it.

Di-graphs. A **di-graph** is another term for a directed graph.

K-partite graphs. In many graphs, we may have a situation where we have two or more groups of nodes, where edges are only allowed between groups and not between nodes of the same group.

Figure A.9 A tri-partite graph. It has 3 partitions, or groups, of nodes: V1, V2, and V3. Within these partitions, edges between nodes are forbidden, but edges are permitted between the partitions.



"Partite" refers to the partitions of node groups, and 'k' refers to the number of those partitions.

In a **mono-partite** graph, there is only one group of nodes and one group of edges. A mono-partite social graph could consist of only "Texan" nodes

connected with “work colleague” edges.

For example, in a social graph, nodes can belong to “New Yorkers” or “Texans” groups, and relationships can belong to “friend” or “work colleague” groups.

A **bi-partite** graph has two node partitions within a graph. Nodes of one group can only connect to nodes of a second type, and not with nodes within their own group. In our social graph example, nodes can belong to “New Yorkers” or “Texans” groups, and relationships can belong to “friend” or “work colleague” groups. In this graph, no New Yorkers would be adjacent to other New Yorkers, and the same for Texans.

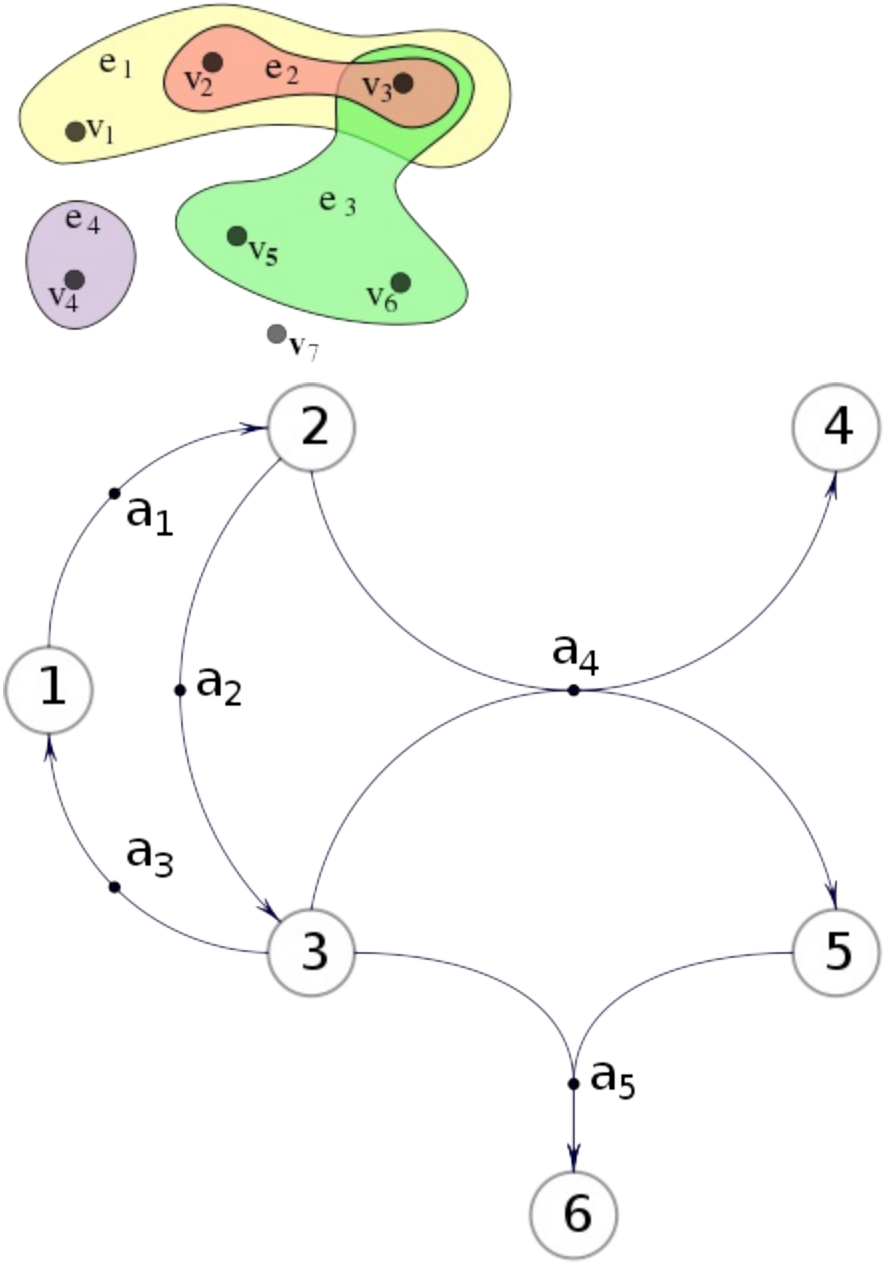
For partitions above three, the requirement that adjacent nodes cannot be the same type still holds. In practice, *k* can be a large number.

Trees. The **tree**, a well studied data structure in machine learning, is a special case of a graph. It is a connected graph without cycles. Another way to describe a graph without cycles is **acyclic**. In the data science and deep learning worlds, a well known example is the directed acyclic graph (DAG), used in designing and governing data workflows.

Hypergraphs. Up to now, our graphs have consisted of edges that connect to two nodes, or one node (a self-loop). For a **hypergraph**, an edge can be incident to more than two nodes. These data structures have a range of applications, including ones that involve the use of GNNs.

Heterogeneous Graphs. A **heterogeneous** graph has multiple node and edge types, while a **multi-relational** graph has multiple edge types.

Figure A.10 Two hypergraphs. On the left, we have an undirected graph whose edges are represented by colored areas, marked *e*, and whose vertices are dots, marked *v*. On the right, we have a directed graph, whose edges are dots, marked *a*, and whose vertices are numbered circles.



A.2 Graph Representations

Now that we have a conceptual idea of what graphs are, we move on to how to express them using the languages of math and code. First, we focus on data structures most relevant to building graph algorithms and storing graph data. We will see that some of these structures, particularly the adjacency matrix, play a prominent role in the GNN algorithms we study in the bulk of this book.

Next, we'll examine a few graph data models. These are important in designing and managing how databases and other other data systems deal with network data.

Lastly, we'll briefly take a look at how graph data is exposed to analysts and engineers via APIs and query languages.

A.2.1 Basic Graph Data Structures

There are a few important ways to represent graphs that can be ported to a computational environment:

- Adjacency matrix - a node-to-node matrix
- Incidence matrix - an edge-to-node matrix
- Edge Lists - a list of edges by their nodes
- Adjacency Lists - Lists of each node's adjacent nodes
- Degree matrix - node-to-node matrix of with degree values
- Laplacian matrix - The degree matrix minus the Adjacency Matrix (**D-A**). Useful in spectral theory

These are by no means the only ways to represent a graph, but from a survey of the literature, software, storage formats and libraries, these seem to be the most prevalant.

In practice, a graph may not be permanently stored as one of these structures, but to execute a needed operation, a graph or sub-graph may be transformed from one representation to another.

What representation is used depends on many factors that should be weighed in planning. These include:

- Size of Graph. How many vertices and edges does the graph contain, and how much are these expected to scale.
- Density of Graph. Is the graph sparse or dense. We'll touch on these terms below.
- Complexity of the Graph's Structure. Is the graph closer to a simple graph, or one that uses one or more of the variations discussed above?
- Algorithms to be used. For a given algorithm, a given data structure may

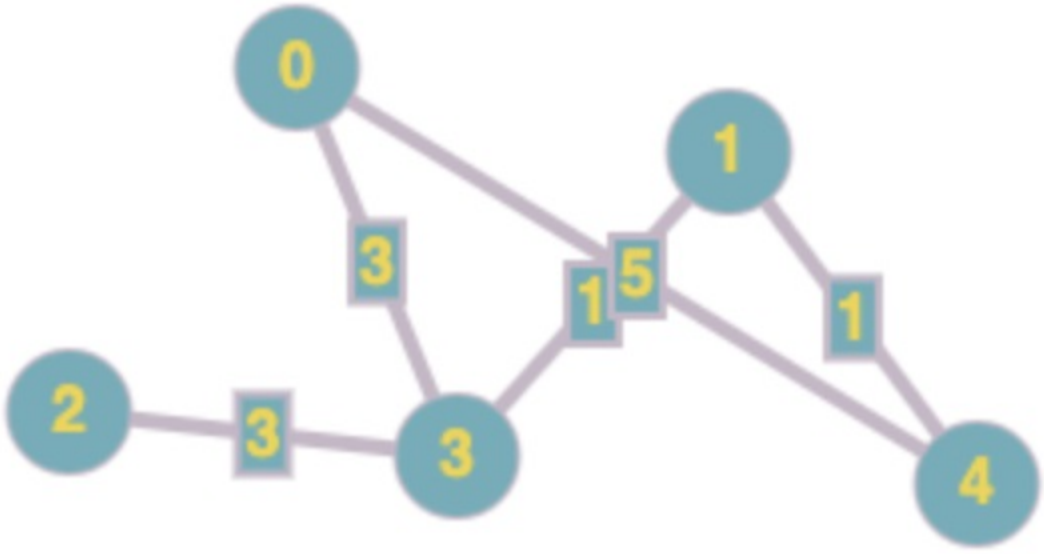
perform relatively weakly or strongly compared to others. Below, for each structure, we'll touch on two simple algorithms to compare.
- Costs to do CRUD operations. How and how frequently will you modify your graph (including **c**reating, **r**eading, **u**pdating, or **d**eleting, nodes, edges, and their attributes) over the course of your operations.

In many data projects, transformation from one data structure to another is common to accommodate particular operations. So, it is normal to employ two or more of the above data structures in a project. In this case, understanding the compute effort to execute the transformation is key. For the most popular structures, graph libraries allow methods that allow seamless transformations, but given the considerations listed above, executing these transformations could take unexpected time or cost.

For the discussion below, we'll talk about how these data structures are used to store topological information about graphs. The only attributes we'll consider are node IDs and edge weights. We'll touch on more properties in section 2.2.3.

To illustrate these concepts, let's use the below weighted graph, consisting of 5 nodes. Circles indicate nodes with their IDs; rectangles are the edge weights.

Figure A.11 An example graph for section A.2.1.



Before we jump in, note that the assessments below apply to these unmodified data structures used to represent simple graphs, as defined above. In practice, people tweak these data structures often to meet their specific requirements. As long as the assumptions and the implications of such tweaks are understood, this shouldn't be a problem.

Adjacency Matrix

An **adjacency matrix** represents a graph in a nxn matrix format. For a graph with *n* nodes, each row or column would describe one node. So, for our example with 5 nodes, we have 5 columns and 5 rows, as you can see in table A.1. These rows and columns are labeled for each node. Cells of the matrix denote adjacency.

Table A.1 An adjacency matrix for the graph in figure A.11.

	0	1	2	3	4

0	0	0	0	3	5
1	0	0	0	1	1
2	0	0	0	3	0
3	3	1	3	0	0
4	5	1	0	0	0

Adjacency matrices can be used for simple directed and undirected graphs. They can also be used for graphs with self loops (where a self-loop would have a value of 1 for the cell that corresponds with a particular single node's column and row).

In an unweighted graph, each cell would either be 0 (no adjacency) or 1 (adjacency). For the diagonal cells, where a single node's column intersects with its respective row, the value would be 0 for a simple graph. For a weighted graph, the values in the cells would be the edge weights. For unweighted parallel edges, the values of the cells would be the number of edges.

For our example, a weighted, undirected graph, the corresponding adjacency matrix is shown above.

Since our graph is undirected, the adjacency matrix is symmetric. For directed graphs, symmetry is possible but not guaranteed.

By inspecting this matrix, we can get a quick visual understanding of the characteristics of the matrix. We can see, for example, how many degrees node 1 has, and get a general idea of the distribution of the degrees. We also see that there are more empty spaces (cells with a 0 value) than edges. This ease of using the matrix to draw quick insights is one advantage of adjacency

matrices.

Adjacency matrices, and matrix representations in general, allow one to analyze graphs by using linear algebra. One relevant example is spectral graph theory (which underlies a few GNN algorithms).

Adjacency matrices are straightforward to implement in python. The matrix in our example can be created using a list of lists, or a numpy array.

```
>>import numpy as np
>>arr = np.array([[0, 0, 0, 3, 3],[0, 0, 0, 1, 1],[0, 0, 0, 3, 0]
```

With our adjacency matrix as a numpy array, let's explore another property of our graph. From our visual inspection of our matrix, we noticed many more zeros than non-zero values. This makes it a sparse matrix. **Sparse matrices,** that is matrices with a large proportion of zero values, can take up unnecessary storage or memory space, and increase calculation times. **Dense matrices**, contrarily, contain a large proportion of non-zero matrices.

What is the sparsity of our matrix?

```
>>sparsity = 1.0 - ( np.count_nonzero(arr) / arr.size )
>>print(sparsity)
>> 0.6
```

So, our matrix has a sparsity of 0.6, meaning 60% of the values in this matrix are zeros.

Inset: Sparsity Using Node Degree

Another way to think about sparsity is in terms of node degree. Let's derive the sparsity value above from the perspective of node degree.

For a simple, undirected graph of n nodes, each node can make at most n-1 connections, and thus have a maximum degree of n-1. The maximum number of edges can be calculated using combinatorics: since each edge represents a pair of nodes, for a set of n nodes, the maximum number of edges is "n choose 2" or (n 2), or N(N-1)/2. For our small matrix, this is 5(5-1)/2 = 10. Thus, defining density as E/N^2, then sparsity can be defined as 1- density. In

our example, this leads to a quantity that agrees with what was calculated using the matrix alone, (1 - 10/25) = 0.6.

Now, think of a graph that has not 5, but millions or billions of nodes. Such graphs exist in the real world, and quite often sparsity can orders of magnitudes less than 0.6. Real world networks can have sparsities in the ranges of 10e-5 and 10e-9 [reference]. For undirected simple graphs, the adjacency matrix is symmetric, so only half the storage is needed. Most of the memory or storage containing the adjacency matrix would be devoted to zero values. Thus, the high sparsity of this data structure leads to memory inefficiencies.

In terms of complexity, for a simple graph, the space complexity would be $\mathbf{O}(n^2)$, for undirected simple graphs. For an undirected graph, due to the symmetry, the space complexity would be $\mathbf{O}(n(n-1)/2)$.

For time complexity, this of course depends on the task or the algorithm. Let's look at two rudimentary tasks, that we'll also address for adjacency list and edge lists:

1. Checking the existence of an edge between a particular pair of nodes.
2. Finding the neighbors of a node.

For the first task, we simply check the row and column corresponding to those nodes. This would take $\mathbf{O}(1)$ time. For the second, we need to check every item in that node's row; this would take $\mathbf{O}(\deg(n))$ time, where deg(n) is the degree of the node.

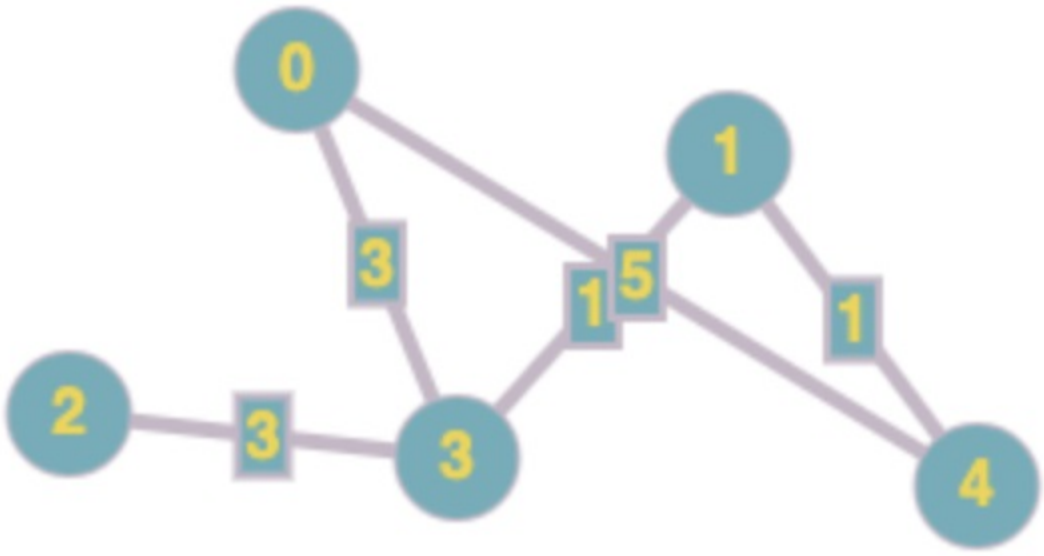
To summarize, advantages of adjacency matrices are that they can quickly check connections between nodes, and are easy to visually interpret. Downsides are that they are less space efficient for sparse matrices. The computational tradeoffs depend on your algorithm. They shine in cases where we have small and dense graphs.

Incidence Matrix

While the adjacency matrix has a row and column for every node, an

incidence matrix represents every edge as a column and every node as a row. Figure A.12 is our example graph again:

Figure A.12 Example graph



Designating the edges in our example by alphabets from left to right, we have our example's incidence matrix in table A.2:

Table A.2 Incidence matrix for the example graph in figure A.13

	A	B	C	D	E
0	0	3	5	0	0
1	0	0	0	1	1

2	3	0	0	0	0
3	3	3	0	1	0
4	0	0	5	0	1

An incidence matrix can represent wider variations of graph types than an adjacency matrix. Multigraphs and hypergraphs are straightforward to express with this data structure.

How does the incidence matrix perform with respect to space and time? To store the data of a simple graph, the incidence matrix has a space complexity of **O**(|E| * |V|), where |*V*| is the number of nodes (*V* for vertices), and |E| is the number of edges. Thus, it is superior to the adjacency matrix for graphs with less edges than nodes, including sparse matrices.

To get an idea of time complexity, we turn to our two simple tasks: checking for an edge, and finding a node's neighbors. To check the existence of an edge, an incidence matrix has a time complexity of O(|E| * |V|), far slower than the adjacency matrix, which does this constant time. To find the neighbors of a node, an incidence matrix also takes O(|E| * |V|).

Overall, incidence matrices have space advantages when used with sparse matrices. For time performance, they have slow performance on the simple tasks we covered. The overall advantage of using incidence matrices are for unambiguously representing complex graphs, such as multigraphs and hypergraphs.

Adjacency Lists

In an **adjacency list**, the aim is to show for each node which vertices it is adjacent. So for *n* nodes, we have *n* lists of neighbors corresponding to each node. Depending on what data structures are used for the lists, properties may also be included in the summary.

For our example, a very simple adjacency list can be shown like this (figure A.13):

Node 0 : Node 3, Node 4

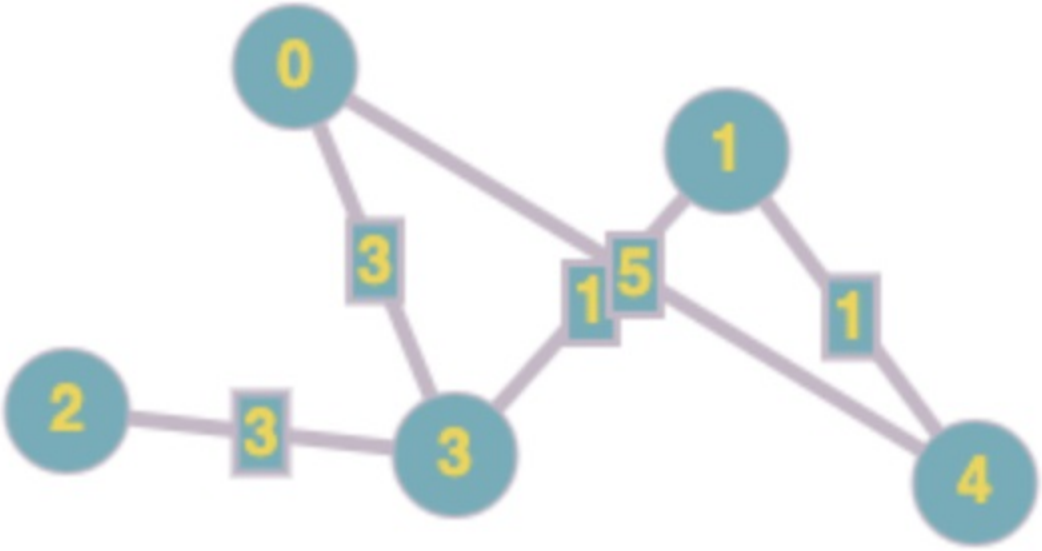
Node 1 : Node 3, Node 4

Node 2 : Node 3

Node 3 : Node 0, Node 1, Node 2

Node 4 : Node 0, Node 1

Figure A.13. Our example graph and its adjacency list.



Such an adjacency list can be accomplished in python using a dictionary with each node as the keys, and lists of the adjacent nodes as values:

{ 0 : [3, 4],

1 : [3, 4],

2 : [3],

3 : [0, 1, 2],

4 : [0, 1] }

We can improve on the dictionary values to allow for the inclusion of the weights of the neighbors:

{ 0 : [(3, 3), (4, 5)],

1 : [(3, 1), (4, 1)],

2 : [(3, 3)],

3 : [(0, 3), (1, 1), (2, 3)],

4 : [(0, 5) , (1, 1)] }

For undirected graphs, the set of nodes doesn't have to be ordered.

Since the adjacency list doesn't devote space to node pairs that are not neighbors, we see that adjacency lists lack the sparsity issues of adjacency matrices. So, to store this data structure, we have a space complexity of **O**(n + v), where *n* is the number of nodes, and *v* is the number of edges.

Going back to the two computational tasks, checking the existence of an edge (task 1) would take **O**(deg(node)) time, where deg(node) is the degree of either node. For this, we simply check every item in that node's list, where worst case we'd have to check them all. For task 2, finding a node's neighbors would also take **O**(deg(node)) time, since we have to inspect every item in that node's list, whose length is the node's degree.

Let's summarize the tradeoffs of an adjacency list. Advantages are that they are relatively efficient in terms of storage, since only edge relationships are stored. This means a sparse matrix would take up less space stored as an adjacency list than an adjacency matrix. Computationally, the tradeoffs depend on the algorithm you are running.

Edge Lists

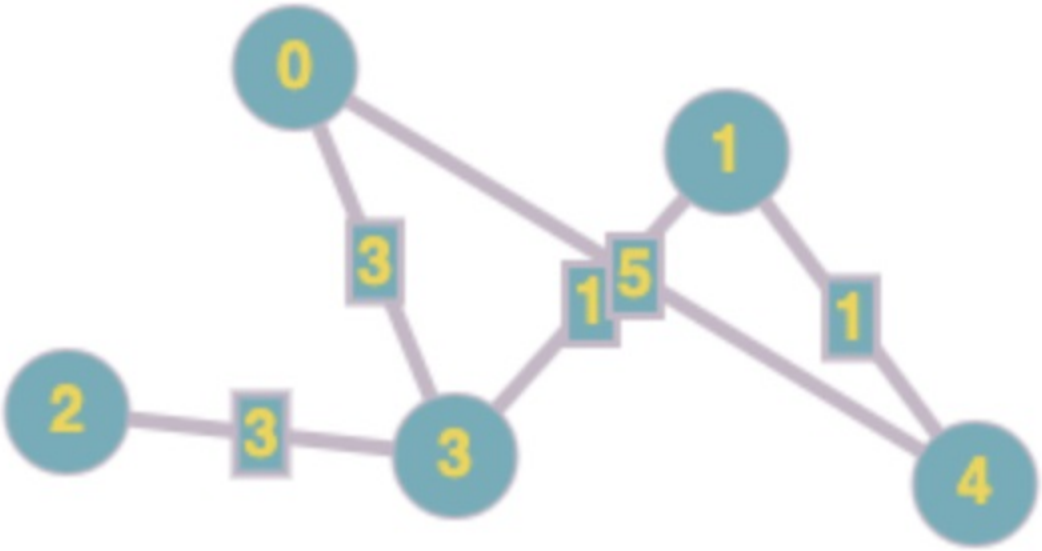
Compared to the preceding two representations, **edge lists** are relatively simple. They consist of a set of doubles (two nodes) or triples (two nodes and an edge weight). These identify a unique edge thusly:

- Node, Node (, Edge Weight) For an undirected graph
- Source Node, Destination Node (, Edge Weight) For a directed graph

Edge lists can represent single, unconnected nodes.

For our example (repeated again as figure A.14):

Figure A.14. Our example graph



the edge list would be:

{ 0, 3, 3 }

{ 0, 4, 5 }

{ 1, 3, 1 }

{ 1, 4, 1 }

{ 2, 3, 3 }

In python, a way to create this would be to use a set of tuples:

```
>> edge_list = {( 0, 3, 3 ), ( 0, 4, 5 ), ( 1, 3, 1 ), ( 1, 4, 1
```

On performance, for storage, the space complexity of an edge list is O(e), where *e* is the number of edges. Regarding our two tasks from above, to establish the existence of a particular edge will have a time complexity of **O**(e), assuming an unordered edge list. To discover all the adjacencies of a node, **O**(e) is the space complexity. In each case, we have to go through the edges in list one by one to check for the edge or the node's neighbor. So, from a compute performance point of view, edge lists have a disadvantage compared to the other two data structures, especially for executing more complex algorithms.

Another advantage of edge lists, from a space point of view, they are more compact than adjacency lists or adjacency matrices. Also, aside from space complexity, they are also simple to create and interpret: it can be a text file where each line only consists of two identifiers separated by a space! For many systems and databases, edge lists in csv or text files are a frequent option to serialize data.

The Laplacian Matrix

One data representation of a graph that is highly valuable in analyzing graphs is the **Laplacian Matrix**. This matrix is key to the development of graph spectral theory, which is in turn critical to the development of spectral based GNN methods.

To produce the laplacian matrix, we subtract the adjacency matrix from the **degree matrix** (**D - A**). The degree matrix is a node-to-node matrix whose values are the degree of a particular node. Here again is our example, (figure A.15):

Figure A.15. Our example graph

The degree matrix for this example is in Table A.3:

Table A.3 Degree matrix for our example graph

Table A.4 Laplacian matrix for our example graph

	0	1	2	3	4
0	2	0	0	0	0
1	0	2	0	0	0
2	0	0	1	0	0
3	0	0	0	3	0
4	0	0	0	0	2

To Laplacian is here:

	0	1	2	3	4
0	2	0	0	-3	-5
1	0	2	0	-1	-1
2	0	0	1	-3	0
3	-3	-1	-3	3	0
4	-5	-1	0	0	2

In practice, laplacian matrices are not used for storage or as a basis for graph operations as the other data structures covered in this section. Their advantages lie in spectral analysis. We discuss spectral graph analysis in later chapters.

A.2.2 Graph Data Models

We are steadily marching from theory to implementation. In the last section, we reviewed common data structures used to represent graphs and their tradeoffs. Graphs can be implemented in these structures from scratch in your preferred programming language, and are also implemented in popular graph processing libraries.

With the listed data structures, we have a variety of ways to implement the structural information in graphs. But graphs and their elements often come with useful attributes and metadata.

A **data model** is an organized way to represent the structural information, attributes, and metadata of a graph. Very much related to this is the notion of a **schema**, which we'll define as a framework that explicitly defines the elements that make up a graph (i.e., nodes, edges, attributes, etc.; also there can be varieties of nodes, edges, etc), and explicitly defines how these

elements work together.

Data models and schemas are critical parts of the scaffolding used to design graph systems such as graph databases and graph processing systems, and are often built upon the data structures reviewed in the last section.

We'll review three such models and provide examples of real systems where they are used.

Minimalist Graph Data Model

The simplest data model uses only nodes, edges, and weights. It can be used on directed or undirected graphs. If weights are used, they can be retrieved using a lookup table.

Pregel, Google's graph processing framework, upon which other popular frameworks are based (including Facebook's Giraph and Apache's GraphX), is based upon such a directed graph. In it, both edges and nodes have an identifier and a single numerical value, which can be interpreted as a weight or attribute.

Property Graph Data Model

In Property Graphs (also called labeled property graphs, or LPGs), allowances are made to confer various metadata to nodes and edges. Such metadata include:

- Identifiers, which distinguish individual nodes and edges
- Labels, which describe classes (or subsets) of nodes or edges.
- Attributes or Properties, which describe individual nodes or edges.

Nodes have an ID, and a set of key/value pairs that can be used to supply additional attributes (also called properties). Similarly, edges have an ID and a set of key/value pairs for attributes.

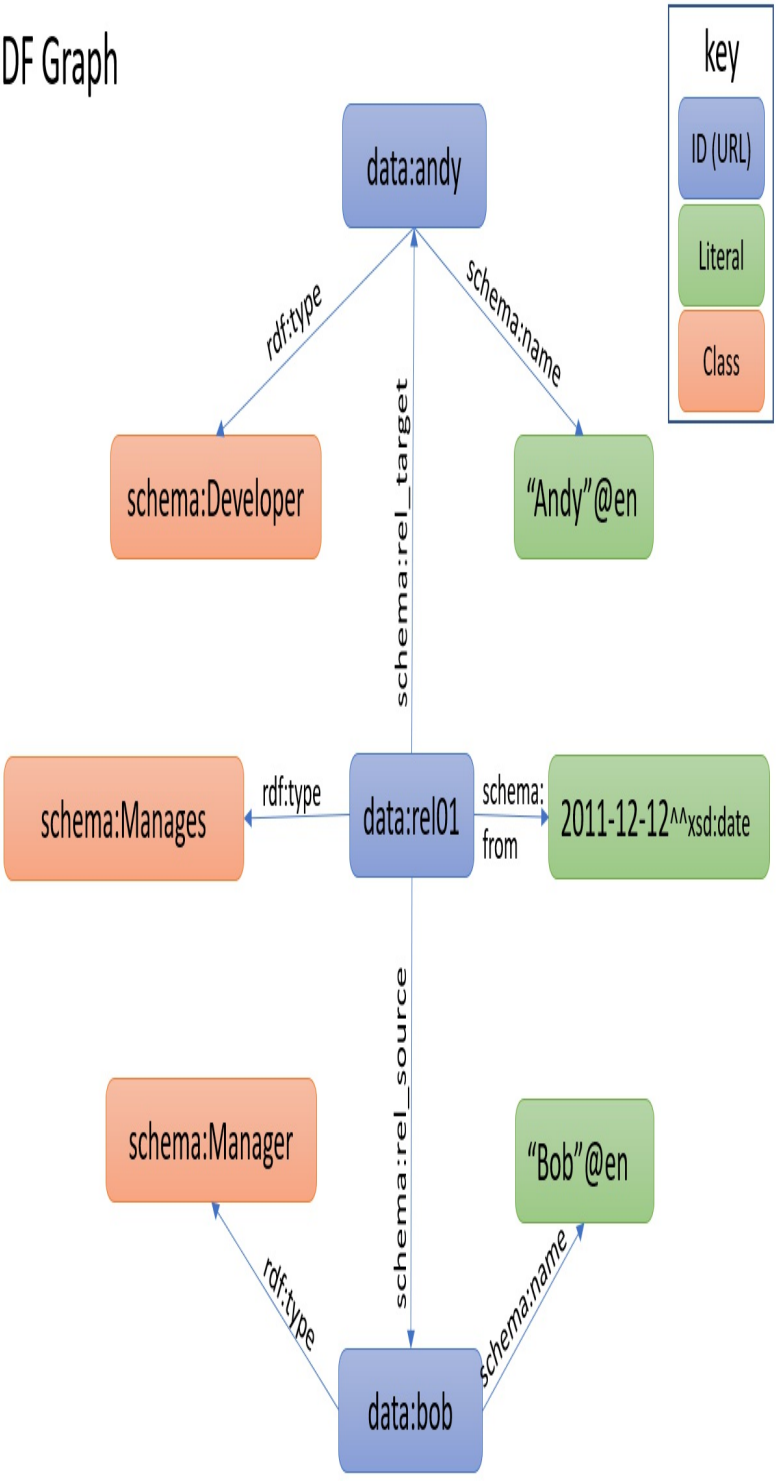
One could think of the property graph as the minimalist graph extended by adding labels, and removing the restrictions on the types and number of

attributes. Here’s a look at a property graph and its equivalent RDF graph (figure A.16):

Figure A.16 Example of a property graph and its equivalent RDF graph.

Labelled Property Graph

RDF Graph



Popular graph databases that use models based on the property graph include Neo4j, Azure's Cosmos, and TigerGraph.

RDF Graph Data Model

RDF (Resource Description Framework; also called Triple Stores) models follow a Subject-Predicate-Object pattern, where Nodes are Subjects and Objects, and Edges are predicates. Nodes and edges have one main attribute, which can be a URI (unique resource identifier) or a literal. URIs, in essense, identify what the 'type' of node or edge being described. Examples of literals can be specific timestamps or dates. Predicates represent relationships, in our example above, the predicate is 'manages'.

Such triples (subject-predicate-object) represent what are called **facts.** Usually facts are directed and flow in the direction from subject to object.

Popular graph databases that use the RDF model include Amazon's Neptune (Neptune also allows the use of LPGs), Virtuoso, and Stardog.

Non Graph Data Model

It should be noted that there are a variety of databases and systems that use neither RDF nor LPG, and store or express nodes, edges and attributes within other storage frameworks, such as document stores, key value stores, and even within a relational database framework.

Knowledge Graphs

There is no unifying definition of a knowledge graph, and this term is used widely in academic, commercial and practitioner circles. Most relevant to GNNs, we define a **knowledge graph** as a representation of knowledge discretized into facts, as defined above. Another way to say this is to define a knowledge graph as a multigraph set onto a specific *subject-relationship-object* schema.

Knowledge graphs may be represented with RDF schemas, but there are

other data models and graph models that can accommodate knowledge graphs.

Here (figure A.17) is an example of a knowledge graph:

Figure A.17 Example of a scifi-themed knowledge graph.

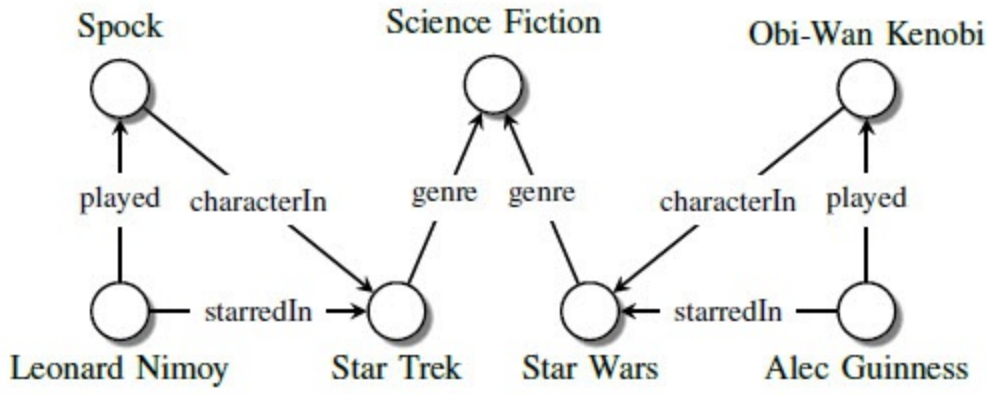


Fig. 1. Sample knowledge graph. Nodes represent entities, edge labels represent types of relations, edges represent existing relationships.

GNN methods are used to embed the data in the nodes and edges, establish the quality of facts, and to discover new entities and relations.

A.2.3 Node and Edge Types

In graphs that have a schema, including knowledge graphs, edges and nodes can be assigned a **type**. Types are part of a defined schema, and as such, govern how data elements interact with each other. They also often have a descriptive aspect.

To distinguish *types* from *properties*, consider that while types help define the ‘rules’ of how data elements work together and how they are interpreted by the data system, properties are descriptive only.

To illustrate types, we can use a road map analogy, where towns are nodes and passages between them are edges. Our edges may include highways, a foot paths, canals, or a bike paths. Each one is a type. Due to geography,

towns can be surrounded by swamps, sit atop mountain peaks, or have other obstacles and impediments to one versus another passage. For towns separated by a desert, passage is only possible by a highway. For other towns, passages can be by multiple passage types. In building this analogy, we see that our town nodes also have types defined by their proximate geography: swamp town, desert town, island town, valley town.

A.2.4 How graphs are exposed

We've talked about data structures and data models to understand how graphs are implemented under the hood. In real life however, most of us won't build graphs from scratch or from the bottom up. When constructing and analyzing graphs, there will be a layer of abstraction between us and the primitive data. In what ways, then, is a graph exposed to the data scientist or engineer? We look at two ways:

- APIs: Using graph libraries or data processing systems
- Query Languages: Querying graph databases via specialized query languages

We briefly explain these, then discuss the graph ecosystem.

APIs: Graph objects in graph systems

When using a graph library or processing software, usually we want the graph we work with to have certain properties and we want to be able to execute operations on the graph. From this lens, it is helpful to think of graphs as software objects that can be operated on by software functions.

In python, an effective way to implement the above is to have a *graph* class, with some operations implemented as methods of the graph class or as stand alone functions. Nodes and edges can be attributes of the graph class, or can have their own *node* and *edge* classes. Properties of graphs implemented in this way, can be attributes of the respective classes.

An example of this is NetworkX, a python-based graph processing library. NetworkX implements a *graph* class. Nodes can be any hashable object,

examples of node objects are integers, strings, files, even functions. Edges are tuple objects of their respective nodes. Both nodes and edges can have properties implemented as python dictionaries.

Below are two short lists of typical methods and attributes of graph classes found in libraries and processing systems.

Basic Methods of Graph Objects

· Graph_Creation - A constructor that creates a new graph object

· **Add_Node, Add_Edge** - Add nodes or edges, and their attributes and labels, if any

· **Get_Node, Get_Edge** - Retrieve stored nodes or edges, with specified attributes and labels

· **Update_Node, Updage_Edge, Update_Graph** - update properties and attributes of nodes, edges and graph objects

· **Delete_Node, Delete_Edge** - Deletes a specified node or edge

Basic Attributes of Graph Objects

· **Number_of_Nodes, Number_of_Edges** - A constructor that creates a new graph object

· **Node_neighbors** - retrieve the adjacent nodes or incident edges of a node

· **Node_List, Edge_List** - Add nodes or edges, and their attributes and labels, if any

· **Connected_Graph** - Retrieve stored nodes or edges, with specified attributes and labels

· **Graph_State** - Retrieve global attributes, labels and properties of the graph

· **Directed_Graph** - Deletes a specified node or edge

Graph query languages

When working with a graph in a graph database, a query language is used. For most relational databases, some variant of SQL is used as the standard language. In the graph database space, there is no standard query language. Below are the languages that currently stand out:

- Gremlin: A language that can be written declaratively or imperatively; designed for database or processing system queries. Developed by the Apache Tinkerpop project and used in several databases (Titan, OrientDB) and processing systems (Giraph, Hadoop, Spark).
- Cypher: A declarative language for property graph-based database queries. Developed by Neo4J, and used by Neo4J and several other databases.
- SPARQL: A declarative query language for RDF-based database queries. Used by Amazon Neptune, Allegrograph, and others.

A.3 Graph systems

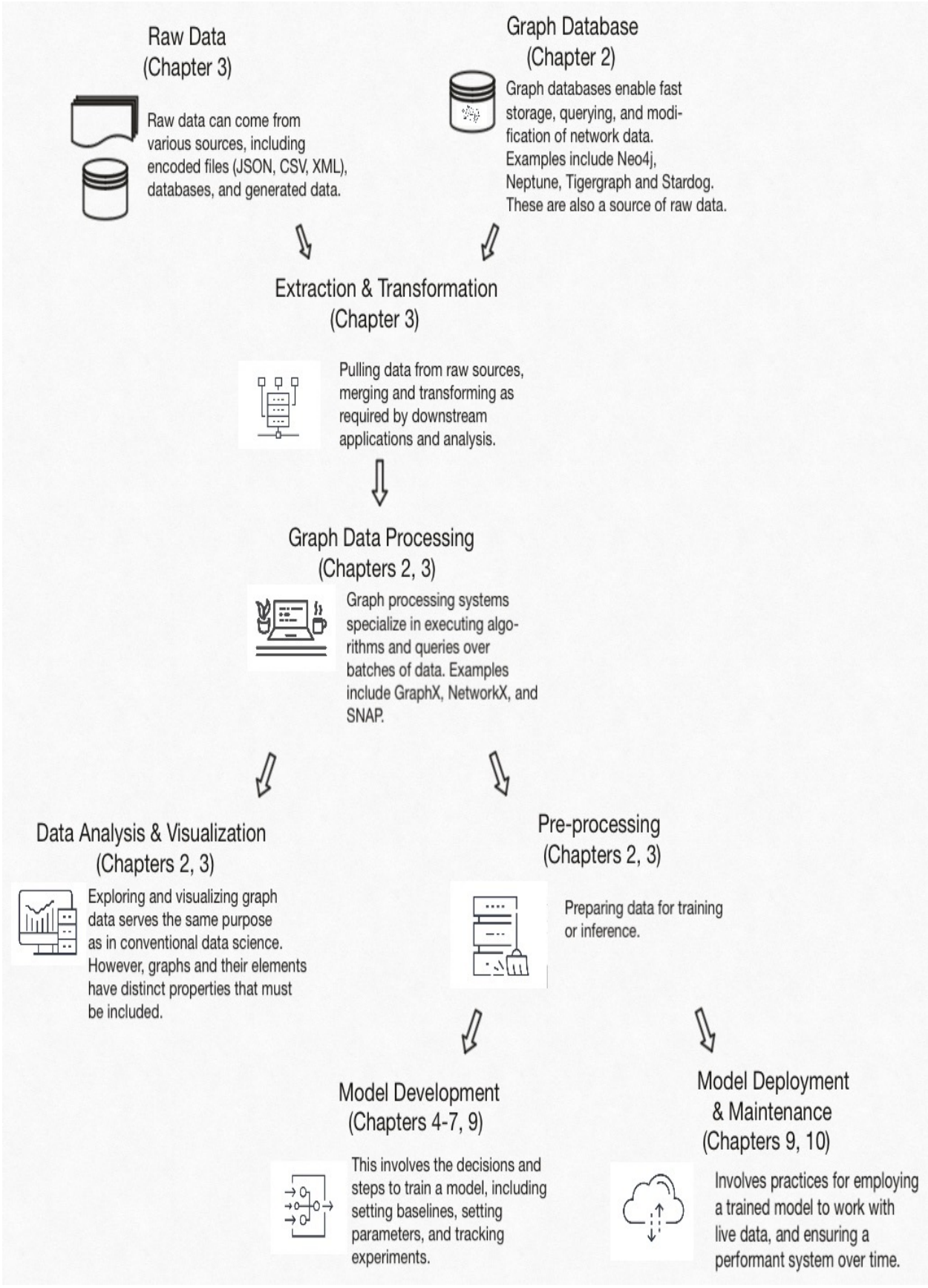
We've covered the basic building blocks that allow us to implement graphs in a programming language. In practice, you will seldom create a graph from scratch, you'll load data into memory or a database using a library or API. The field of graph libraries, databases, and commercial software is broad and growing rapidly. A good way to decide upon what to use is to start with your use case and requirements, then choose your development and deployment architecture from there. This section will briefly give an overview of this landscape to help you. The taxonomy we will develop below is by no means absolute, but should serve a useful guideline.

A.3.1 Graph workflow

In chapter 1, we touched on the GNN workflow. In this section, we'll delve more deeply into it. The workflow or pipeline leading to the development of a GNN mirrors directly that of a machine learning workflow. The key differences are in the details of the broad steps. Typically, we use a set of different tools at each step for graph-related tasks than we do for euclidean

datasets. Figure A. 18 shows this workflow:

Figure A.18 A graph data workflow.


Raw Data
(Chapter 3)
Raw data can come from various sources, including encoded files (JSON, CSV, XML), databases, and generated data.
Graph Database
(Chapter 2)
Graph databases enable fast storage, querying, and modification of network data. Examples include Neo4j, Neptune, Tigergraph and Stardog. These are also a source of raw data.
Extraction & Transformation
(Chapter 3)
Pulling data from raw sources, merging and transforming as required by downstream applications and analysis.
Graph Data Processing
(Chapters 2, 3)
Graph processing systems specialize in executing algorithms and queries over batches of data. Examples include GraphX, NetworkX, and SNAP.
Data Analysis & Visualization
(Chapters 2, 3)
Exploring and visualizing graph data serves the same purpose as in conventional data science. However, graphs and their elements have distinct properties that must be included.
Pre-processing
(Chapters 2, 3)
Preparing data for training or inference.
Model Development
(Chapters 4-7, 9)
This involves the decisions and steps to train a model, including setting baselines, setting parameters, and tracking experiments.
Model Deployment
& Maintenance
(Chapters 9, 10)
Involves practices for employing a trained model to work with live data, and ensuring a performant system over time.

In this appendix, we will focus on data extraction, transformation, and exploration.

ETL or Extract-Transform-Load is the step that has to do with collecting or creating your raw data, making some optional transformations that are suitable for storage, and then moving that transformed data into a storage system.

ETL with graph data may be as simple as importing an edge list into your graph database with built-in commands. Or it may entail programmatically combining data from multiple sources, then storing that combined data into an adjacency list, then transforming that into a property graph schema for storage in a graph database.

Because of these and other complexities, schema design is important.

ETL capabilities are typically built into graph databases and frameworks.

EDA or Exploratory Data Analysis involves using summary statistics, metrics, and visualizations to get a grasp of the relevant characteristics of a given dataset. Such insights inform the next steps of a machine learning project, including what tools to use downstream, and what models and DL architectures to apply. EDA also helps highlight potential problems that may occur downstream in the workflow. EDA was covered in more detail in the last section.

It should be noted that at every data transformation step, some EDA could be included. So, even after preprocessing, EDA could be done.

Preprocessing is a step that involves further transformation of the data in order to apply a particular algorithm, including model training, to a dataset. The starting point here is not raw data, but data that has gone through the ETL step above, and has a home in a database or is serialized.

Preprocessing can involve filtering data, creating features, segmenting the data, annotating data, and any number of methods.

Algorithms including Model Training are what the pre-processed data feeds into. Most of this book takes a detailed examination of various GNN algorithms.

Model Deployment involves the steps after a model has been successfully trained and optimized for performance. There are many aspects of deployment that are beyond the scope of this book. Issues addressed in later chapters include dealing with GNNs at scale, and ethics in GNNs.

A.3.2 Graph ecosystem

Given the workflow above, what are specific tools and how are they segmented. At the time of writing, commercial and open source tools for graph analysis, ML modeling, visualization and storage are expanding relatively rapidly. There is quite a lot of overlap between tools and functions, and many hybrid tools that don't neatly fit into any category, so that there is no clean delineation of segments. So, the approach for this chapter will be to highlight basic segments, and focus on the most popular tools.

We'll focus on the following segments:

- Graph Frameworks or Graph Compute Engines
- Graph Databases
- Visualization Libraries
- GNN Libraries

Scope: When categorizing data tools, another way of partitioning is by the scope of data which is targeted. This can be a narrow view, whose focus is on one record or a small set of records. OLTP or Online Transactional Processing embodies this view. Financial transactions of a bank is an example. Let's say that for a very small bank with a simple database, each row in a databases would represent an individual account. When individuals deposit funds, make purchases, or check their online accounts, in this simple database, these actions would only impact one record. An OLTP data query could be "how much money does Bob have right now?" or "How much money was withdrawn from Bob's account over the last week?"

For OLAP (Online Analytical Processing), the scope is on analyzing many

records at once. These systems are optimized for queries that examine batches of data. For the example above, the data system would be fielding queries across many individual accounts, say for every account at Bob's bank, instead of just Bob. Such queries may be: "For the entire bank, how many accounts withdrew money two weeks ago?" or "How much money was deposited today by every bank account holder?"

Graph Databases. Graph databases are the graph analogues of traditional relational databases from a functional standpoint. Such databases were devised to handle OLTP-focused transactions. They allow CRUD (create, read, update, and delete) transactions. They also tend to follow ACID (atomicity, consistency, isolation, and durability) principles regarding the integrity of the data. Graph databases of this type differ from relational databases in that they store data using graph data models and schemas. At the time of writing, the most popular graph databases are Neo4j, Microsoft Cosmos DB, OrientDB, and ArangoDB. Except for Neo4J, these databases support multiple models including property graphs. Neo4j supports property graphs only. The most popular databases that support RDF models are Virtuoso and Amazon Neptune.

In addition to property graph and RDF databases, other types of non-graph databases are used to store graph data. Document stores, relational databases, and key-value stores are examples. To utilize such non-graph databases with graph data models, one must carefully define how the existing schema maps to the graph elements and their attributes.

Graph Compute Engines (or Graph Frameworks). Graph Compute Engines are designed to make queries using batches of data. Such queries can output aggregate statistics, or output graph-specific items, such as cluster identification and find shortest paths. These data systems tend to follow the OLAP model. It is not unusual for such systems to work closely with a graph database, which serves the input data batches needed for the analytic queries. Examples of such systems are Spark's GraphX, Giraph, Stanford's SNAP.

Visualization Systems. Graph visualization tools share characteristics with graph compute engines, as they are geared towards analytics vs transactional queries and computations. However, such tools are designed to create aesthetic and useful images of the networks under analysis. In the best

visualization tools, these images are interactive and dynamic. Outputs of visualization systems can be optimized for presentation on the web, or in printed format with high definition. Examples of such tools are Gephi, Cytoscape, and Tulip.

Graph Representation and Neural Network Libraries. The last segment of graph tools are the central subject of this book. I'm bucketing software tools that create graph embeddings with SW that allows the training of learning models based on graph data. At the time of writing, there are many solutions available. I will focus on the most visible ones in the main text, and list a larger amount of tools in the appendix. I'll summarize this segment below, as the next chapter will be dedicated to this topic.

Graph representation tools range from dedicated, stand-alone libraries (Pytorch BigGraph) to graph systems that have embedding as a feature (Neo4j as a database and SNAP as a compute framework).

Graph Neural Network libraries come as standalone libraries, and as libraries that use Tensorflow or Pytorch as a backend. In this text, the focus will be on Pytorch Geometric. Other popular libraries include DGL (Deep Graph Library, a standalone library) and Spektral (which uses Kera and Tensorflow as a backend). The best libraries will not only implement a range of deep learning layers, but have benchmark datasets.

A.4 Graph algorithms

As the field of graphs has been around for a while, graph algorithms, algorithms that are based on a graph data structure, have proliferated in that time.

Having an understanding of well-used graph algorithms can provide valuable context with which to think about the algorithms used in neural networks. Graph algorithms can also serve as sources of node, edge, or graph features for machine learning. Such features can be a point of comparison with the features generated by GNNs. Finally, as with machine learning in general, sometimes a statistical model is not the best solution. Having an understanding of the analytical landscape can help when deciding whether or

not to use a GNN solution.

In this section, we review two types of graph algorithms. We provide a general description, explaining why they are important. For an in-depth treatment on this topic, review the references at the end of the chapter, particularly Cormen, Deo, and Skiena.

As stated in section A.2.1, the performance of graph algorithms heavily depends upon the choice of data structure.

Traversal and Search Algorithms. In section A.1.1, we discussed the concept of a walk and a path. In these fundamental concepts, we get from one node in a graph to another by traversing a set of nodes and edges between them.

For large graphs with many non-unique walks and paths between node pairs, how do we decide which path to take? And for graphs we haven't explored and don't have a 'map' of, what is the best way to create that map? Wrapped into these questions is the issue of what direction to take when traversing a graph at a particular node. For a node of degree 1, this answer is trivial; for a node with degree 100, the answer is less so.

Traversal algorithms offer systematic ways to walk a graph. For such algorithms, we start at a node, and following a set of rules, we decide upon the next node to which to hop. Often, as we conduct the walk, we keep track of nodes and edges that have been encountered. For certain algorithms, if we outline the path taken, we can end up with a tree structure.

Three well known strategies for traversal are:

- Breadth first: A breadth-first traversal prefers to explore all of the immediate neighbors of a node before going further away. This is also known as breadth first search (BFS).
- Depth first: In depth-first search (DFS), rather than explore every immediate neighbor first, we follow each new node without regard to its relationship to the current node. This is done in such a way that every node is encountered at least once and every edge is encountered exactly once.
- There are versions of DFS and BFS for directed graphs

- Random: In random traversals, in contrast to BFS and DFS, where traversal is governed by a set of rules, traversal to the next node is done randomly. For a starting node of degree 4, in a random traversal with a uniform distribution, each neighboring node would have a 25% chance to be chosen. Such methods are used in algorithms like DeepWalk and Node2Vec (covered in chapter 3).

Shortest Path. An enduring problem highly related to graphs is that of the shortest path. Interest in solving this problem has existed for decades (a great survey paper of shortest path methods was published as far back as 1969), with several distinct algorithms existing. Modern applications of shortest path methods are used in navigation applications, like finding the fastest route to a destination.

Variations of such algorithms include:

- Shortest path between:
  - Two nodes
  - Two nodes on a path that includes specified nodes
  - All nodes
  - One node to all others
- Ranked shortest paths (i.e., second shortest path, third shortest, etc)

Such algorithms can also take into account weights in graphs. In these cases, shortest path algorithms are also called least cost algorithms.

A highly lauded algorithm for least cost determination is Dijkstra's (pronounced DYKE-strə) Algorithm. Given a node, it finds the shortest path to every other node or to a specified node. As this algorithm progresses, it traverses the graph while keeping track of the distance and connecting nodes (to the start node) of each node it encounters. It prioritizes the nodes encountered by their shortest (or least cost) path to the start node. As it traverses, it prioritizes low cost paths.

A.5 How to read graph literature

GNNs are a rapidly proliferating topic. New methods and techniques have

been proposed in a short span of time. Though this book focuses on practical and commercial applications of graphs, much of the state of the art in this field is disclosed in academic journals and conferences. Knowing how to effectively study publications from these sources is essential to keep up to speed with the field and to encounter valuable ideas that can be implemented in code.

In this short section, we list some commonly used notations to describe graphs in technical publications.

A few tips on reading academic literature as a practitioner, someone focused on using the methodology in the paper to add value to a project that has time constraints:

- To efficiently extract value from a paper, one must be selective on which sections of the publication to focus on. One should focus on clearly understanding the problem statement and understanding the solution that can be translated into code. This sounds obvious, but many papers include sections that for a practitioner are distracting at best. Mathematical proofs and long historical notes are examples.
- While a positive trend is that papers are starting to make reproducibility easier with included code and data, it may not be possible to reproduce a paper for one reason or another. That's why it's important to reach out to the authors if you think something is missing or doubtful.
- Look closely at indicators of the application scope of the problem and solution. An exciting development may not be applicable to your problem, and it may not be immediately obvious.

In the vast majority of papers, ideas are built around set theory and linear algebra, using concepts like sets, vectors, matrices and tensors to describe graphs and their elements. In describing algorithms, including deep learning algorithms, operations between these mathematical entities are also used.

Common Graph Notations

In mathematical notation, a graph is described as a set of nodes and edges:

G = (V, E)

Where *V* and *E* are collections or sets of vertices (nodes) and edges, respectively. When we want to express the count of elements in these collections, we use |*V*| and |*E*|.

For directed graphs, an accented G is sometimes, but not always used: 
.

Individual nodes and edges are denoted by lower case letters, *v* and *e*, respectively.

When referring to a pair of adjacent nodes, we use *u* and *v*. Thus, an edge can also be expressed as {*u*, *v*}, or *uv*.

When dealing with weighted graphs, a weight for a particular edge is expressed as *w*(*e*). In terms of an edge's nodes, we can include the weight as {*u*, *v*, *w*}.

To express the features of a graph or its elements, we use the notation *x* or **x** when the features are expressed as a vector or matrix, respectively.

For graph representations, since many such representations are matrices, bold letters are used to express them: **A** for the adjacency matrix, **L** for the Laplacian matrix, and so on.

A.6 Summary

- Graphs are data structures that have a simple basis (nodes and edges), but have widely varying and complex implementations.
- Graphs and their elements can be described and compared using a set of properties that are common across applications.
- For analytical and computational purposes, there are a few ways to represent a graph. The chosen representation has implications for storage, computational performance, and human interpretability.

- There is a basic workflow for graph analytics and graph learning that follows a few common tasks and stages.
- There is an ecosystem of graph tools, libraries, commercial software, and databases that is used in a graph workflow. These tools have tradeoffs that must be considered.
- Understanding basic graph algorithms is important in analyzing graphs and in understanding how more complex algorithms, like the ones used to build GNNs work.
- In a rapidly advancing field, an important source of ideas and learning is from published research and other graph-related literature.

A.7 References

Besta, M., et. al. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. ArXiv, abs/1910.09017, 2019.

Cormen, T; et. al. Introduction to Algorithms, MIT Press, 3rd Edition, 2009.

Dreyfus, S.E., An Appraisal of Some Shortest Path Algorithms. Operations Research, 1969.

Deo, Narsingh, Graph Theory with Applications to Engineering and Computer Science. Dover Books on Mathematics, 2017.

Duong, et al, On Node Features for Graph Neural Networks.

Fensel, D, et. al. Knowledge Graphs, Methodology, Tools, and Selected Use Cases. Springer 2020.

Goodrich, M.; Tamassia, R., Algorithm Design and Applications, Wiley, 2015.

Hamilton, W. Graph Representation Learning, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool, 2020.

Skiena, S., The Algorithm Design Manual. Springer-Verlag, 1997.

Nickel, M, A Review of Relational Machine Learning for Knowledge Graphs, arXiv:1503.00759v3, 2015.

Resource Description Framework (RDF): Concepts and Abstract Syntax, W3C Recommendation 10 February 2004, https://www.w3.org/TR/rdf-concepts/.