An algorithmic reasoning approach to GNNs

Angela Carraro and Matteo Scorcia

DSSC + IN20 - UniTS

Contents

Chapter 1

Graph Representation Learning

Summary of [6].

1.1 Introduction

Graphs are a widespread data structure and a universal language for describing complex systems. In the most general view, a graph is simply a collection of objects (i.e., nodes), along with a set of interactions (i.e., edges) between pairs of these objects.

The power of the graph formalism lies both in its focus on *relationships between points* (rather than the properties of individual points), as well as in its generality.

1.1.1 What is a graph?

Before we discuss machine learning on graphs, it is necessary to give a bit more formal description of what exactly we mean by "graph data". Formally, a *graph* $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes. We denote an edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$. In many cases we will be concerned only with *simple graphs*, where there is at most one edge between each pair of nodes, no edges between a node and itself, and where the edges are all undirected, i.e., $(u, v) \in E \iff (v, u) \in E$.

graph

A convenient way to represent graphs is through an *adjacency matrix* $\mathbf{A} \in \mathbb{R}^{|V| \times |V|}$. To represent a graph with an adjacency matrix, we order the nodes in the graph so that every node indexes a particular row and column in the adjacency matrix. We can then represent the presence of edges as entries in

adjacency matrix

this matrix: $\mathbf{A}[u, v] = 1$ if $(u, v) \in E$ and $\mathbf{A}[u, v] = 0$ otherwise. If the graph contains only undirected edges then $\mathbf{A}$ will be a symmetric matrix, but if the graph is directed (i.e., edge direction matters) then $\mathbf{A}$ will not necessarily be symmetric. Some graphs can also have weighted edges, where the entries in the adjacency matrix are arbitrary real-values rather than $\{0, 1\}$.

Beyond the distinction between undirected, directed and weighted edges, we will also consider graphs that have different types of edges. In these cases we can extend the edge notation to include an edge or relation type τ, e.g., $(u, \tau, v) \in E$, and we can define one adjacency matrix $\mathbf{A}_\tau$ per edge type. We call such graphs *multi-relational*, and the entire graph can be summarized by an adjacency tensor $\mathcal{A} \in \mathbb{R}^{|V| \times |\mathcal{R}| \times |V|}$, where $\mathcal{R}$ is the set of relations.

multi-relational graph

Lastly, in many cases we also have *attribute* or *feature* information associated with a graph. Most often these are node-level attributes that we represent using a real-valued matrix $\mathbf{X} \in \mathbb{R}^{|V| \times m}$, where we assume that the ordering of the nodes is consistent with the ordering in the adjacency matrix. In some cases we even associate real-valued features with entire graphs.

1.1.2 Machine learning on graphs

Node classification

node classification

In *node classification* the goal is to predict the label y_u — which could be a type, category, or attribute — associated with all the nodes $u \in V$, when we are only given the true labels on a training set of nodes $V_{\text{train}} \subset V$. Node classification is perhaps the most popular machine learning task on graph data, especially in recent years.

nodes not i.i.d.

At first glance, node classification appears to be a straightforward variation of standard supervised classification, but there are in fact important differences. The most important difference is that the nodes in a graph are not *independent and identically distributed (i.i.d.)*. Usually, when we build supervised machine learning models we assume that each datapoint is statistically *independent* from all the other datapoints; otherwise, we might need to model the dependencies between all our input points. We also assume that the datapoints are *identically distributed*; otherwise, we have no way of guaranteeing that our model will generalize to new datapoints. Node classification completely breaks this i.i.d. assumption. Rather than modeling a set of i.i.d. datapoints, we are instead modeling an interconnected set of nodes.

homophily

structural equivalence

In fact, the key insight behind many of the most successful node classification approaches is to explicitly leverage the connections between nodes. One particularly popular idea is to exploit *homophily*, which is the tendency for nodes to share attributes with their neighbors in the graph. Based on the notion of homophily we can build machine learning models that try to assign similar labels to neighboring nodes in a graph. Beyond homophily there are also concepts such as *structural equivalence*, which is the idea that nodes with similar local

neighborhood structures will have similar labels, as well as *heterophily*, which presumes that nodes will be preferentially connected to nodes with different labels. For example, gender is an attribute that exhibits heterophily in many social networks (boys want to connect to girls and vice versa). When we build node classification models we want to exploit these concepts and model the relationships between nodes, rather than simply treating nodes as independent datapoints.

heterophily

Supervised or semi-supervised? Due to the atypical nature of node classification, researchers often refer to it as semi-supervised. This terminology is used because when we are training node classification models, we usually have access to the full graph, including all the unlabeled (e.g., test) nodes. The only thing we are missing is the labels of test nodes. However, we can still use information about the test nodes (e.g., knowledge of their neighborhood in the graph) to improve our model during training. This is different from the usual supervised setting, in which unlabeled datapoints are completely unobserved during training.

The general term used for models that combine labeled and unlabeled data during training is semi-supervised learning, so it is understandable that this term is often used in reference to node classification tasks. It is important to note, however, that standard formulations of semi-supervised learning still require the i.i.d. assumption, which does not hold for node classification. Machine learning tasks on graphs do not easily fit our standard categories!

Edge prediction

Node classification is useful for inferring information about a node based on its relationship with other nodes in the graph. But what about cases where we are missing this relationship information? Can we use machine learning to infer the edges between nodes in a graph?

This task goes by many names, such as link prediction, graph completion, and relational inference, depending on the specific application domain. We will simply call it *edge prediction* here. Along with node classification, it is one of the more popular machine learning tasks with graph data and has countless real-world applications: recommending content to users in social platforms, predicting drug side-effects, or inferring new facts in a relational databases — all of these tasks can be viewed as special cases of relation prediction.

edge prediction

The standard setup for relation prediction is that we are given a set of nodes V and an incomplete set of edges between these nodes $E_{\text{train}} \subset E$. Our goal is to use this partial information to infer the missing edges $E\ E_{\text{train}}$. Like node classification, relation prediction blurs the boundaries of traditional machine learning categories — often being referred to as both supervised and unsupervised — and it requires inductive biases that are specific to the graph domain.

Clustering and community detection

community detection

Both node classification and relation prediction require inferring *missing* information about graph data, and in many ways, those two tasks are the graph analogues of supervised learning. *Community detection*, on the other hand, is the graph analogue of unsupervised clustering. The challenge of community detection is to infer latent community structures given only the input graph $G = (V, E)$.

Graph classification, regression, and clustering

graph classification or regression

graph clustering

In these *graph classification or regression* applications, we seek to learn over graph data, but instead of making predictions over the individual components of a single graph (i.e., the nodes or the edges), we are instead given a dataset of *multiple different graphs* and our goal is to make independent predictions specific to each graph. In the related task of *graph clustering*, the goal is to learn an unsupervised measure of similarity between pairs of graphs.

Of all the machine learning tasks on graphs, graph regression and classification are perhaps the most straightforward analogues of standard supervised learning. Each graph is an i.i.d. datapoint associated with a label, and the goal is to use a labeled set of training points to learn a mapping from datapoints (i.e., graphs) to labels. In a similar way graph clustering is the straightforward extension of unsupervised clustering for graph data. The challenge in these graph-level tasks, however, is how to define useful features that take into account the relational structure within each datapoint.

1.2 Background and Traditional Approaches

Before we introduce the concepts of graph representation learning and deep learning on graphs, it is necessary to give some methodological background and context. What kinds of methods were used for machine learning on graphs prior to the advent of modern deep learning approaches? In this section, we will provide a very brief and focused tour of traditional learning approaches over graphs. This background chapter will also serve to introduce key concepts from *graph analysis* that will form the foundation for later sections.

1.2.1 Graph Statistics and Kernel Methods

Node-level statistics and features

What are useful properties and statistics that we can use to characterize the nodes in this graph?

In principle the properties and statistics we discuss below could be used as

features in a node classification model (e.g., as input to a logistic regression model). We will consider some features that could be used to differentiate the nodes in a network, and the properties we will discuss are generally useful across a wide variety of node classification tasks.

Node degree

The most obvious and straightforward node feature to examine is the *degree*, which is usually denoted d_u for a node $u \in V$ and simply counts the number of edges incident to a node:

degree

$$d_u = \sum_{v \in V} \mathbf{A}[u, v]. \tag{1.1}$$

Note that in cases of directed graphs, one can differentiate between different notions of degree — e.g., corresponding to outgoing edges or incoming edges by summing over rows or columns in Equation 1.1. In general, the degree of a node is an essential statistic to consider, and it is often one of the most informative features in traditional machine learning models applied to node-level tasks.

Node centrality

Node degree simply measures how many neighbors a node has, but this is not necessarily sufficient to measure the *importance* of a node in a graph. In many cases we can benefit from additional and more powerful measures of node importance. To obtain a more powerful measure of importance, we can consider various measures of what is known as *node centrality*, which can form useful features in a wide variety of node classification tasks.

node centrality

One popular and important measure of centrality is the so-called *eigenvector centrality*. Whereas degree simply measures how many neighbors each node has, eigenvector centrality also takes into account how important a node's neighbors are. In particular, we define a node's eigenvector centrality e_u via a recurrence relation in which the node's centrality is proportional to the average centrality of its neighbors:

eigenvector centrality

$$e_u = \frac{1}{\lambda} \sum_{v \in V} \mathbf{A}[u, v] e_v \qquad \forall u \in V, \tag{1.2}$$

where λ is a constant. Rewriting this equation in vector notation with $\mathbf{e}$ as the vector of node centralities, we can see that this recurrence defines the standard eigenvector equation for the adjacency matrix:

$$\lambda \mathbf{e} = \mathbf{A}\mathbf{e}. \tag{1.3}$$

In other words, the centrality measure that satisfies the recurrence in Equation 1.2 corresponds to an eigenvector of the adjacency matrix. Assuming that we require positive centrality values, we can apply the Perron-Frobenius Theorem — for our purposes the key assertion in the theorem is that any irreducible square matrix has a unique largest real eigenvalue, which is the only eigenvalue whose corresponding eigenvector can be chosen to have strictly positive components— to further determine that the vector of centrality values $\mathbf{e}$ is given by the eigenvector corresponding to the largest eigenvalue of $\mathbf{A}$.

One view of eigenvector centrality is that it ranks the likelihood that a node is visited on a random walk of infinite length on the graph. This view can be illustrated by considering the use of power iteration to obtain the eigenvector centrality values. That is, since λ is the leading eigenvector of $\mathbf{A}$, we can compute $\mathbf{e}$ using power iteration via

$$\mathbf{e}^{(t+1)} = \mathbf{A}\mathbf{e}^{(t)}. \tag{1.4}$$

Note that we have ignored the normalization in the power iteration computation for simplicity, as this does not change the main result. If we start off this power iteration with the vector $\mathbf{e}^{(0)} = (1, 1, \ldots, 1)^T$, then we can see that after the first iteration $\mathbf{e}^{(1)}$ will contain the degrees of all the nodes. In general, at iteration $t \geq 1$, $\mathbf{e}^{(t)}$ will contain the number of length-t paths arriving at each node. Thus, by iterating this process indefinitely we obtain a score that is proportional to the number of times a node is visited on paths of infinite length. This connection between node importance, random walks, and the spectrum of the graph adjacency matrix will return often throughout the ensuing sections and chapters of this book.

The clustering coefficient

clustering coefficient

The *clustering coefficient* measures the proportion of closed triangles in a node's local neighborhood. The popular *local variant* of the clustering coefficient is computed as follows:

$$c_u = \frac{|(v_1, v_2) \in E : v_1, v_2 \in \mathcal{N}(u)|}{\binom{d_u}{2}}. \tag{1.5}$$

The numerator in this equation counts the number of edges between neighbours of node u (where we use $\mathcal{N}(u) = \{v \in V : (u, v) \in E\}$ to denote the node neighborhood). The denominator calculates how many pairs of nodes there are in u's neighborhood. The clustering coefficient takes its name from the fact that it measures how tightly clustered a node's neighborhood is. A clustering coefficient of 1 would imply that all of u's neighbors are also neighbors of each other.

Graph-level features and graph kernels

So far we have discussed various statistics and properties at the node level, which could be used as features for node-level classification tasks. However, what if our goal is to do graph-level classification? Many of the methods we survey here fall under the general classification of *graph kernel methods*, which are approaches to designing features for graphs or implicit kernel functions that can be used in machine learning models.

Bag of nodes

The simplest approach to defining a graph-level feature is to just aggregate node-level statistics. For example, one can compute histograms or other summary statistics based on the degrees, centralities, and clustering coefficients of

the nodes in the graph. This aggregated information can then be used as a graph-level representation. The downside to this approach is that it is entirely based upon local node-level information and can miss important global properties in the graph.

The Weisfeiler-Lehman kernel

One way to improve the basic bag of nodes approach is using a strategy of *iterative neighborhood aggregation*. The idea with these approaches is to extract node-level features that contain more information than just their local ego graph, and then to aggregate these richer features into a graph-level representation.

Perhaps the most important and well-known of these strategies is the WeisfeilerLehman (WL) algorithm and kernel.

1.2.2 Neighborhood Overlap Detection

In the last section we covered various approaches to extract features or statistics about individual nodes or entire graphs. These node and graph-level statistics are useful for many classification tasks. However, they are limited in that they do not quantify the *relationships* between nodes. For instance, the statistics discussed in the last section are not very useful for the task of relation prediction, where our goal is to predict the existence of an edge between two nodes.

In this section we will consider various statistical measures of neighborhood overlap between pairs of nodes, which quantify the extent to which a pair of nodes are related. For example, the simplest *neighborhood overlap measure* just counts the number of neighbors that two nodes share:

neighborhood overlap measure

$$\mathbf{S}[u, v] = |\mathcal{N}(u) \cap \mathcal{N}(v)|, \tag{1.6}$$

where we use $\mathbf{S}[u, v]$ to denote the value quantifying the relationship between nodes u and v and let $\mathbf{S} \in \mathbb{R}^{|V| \times |V|}$ denote the *similarity matrix* summarizing all the pairwise node statistics.

similarity matrix

Even though there is no "machine learning" involved in any of the statistical measures discussed in this section, they are still very useful and powerful baselines for relation prediction. Given a neighborhood overlap statistic $\mathbf{S}[u, v]$, a common strategy is to assume that the likelihood of an edge (u, v) is simply proportional to $\mathbf{S}[u, v]$:

$$\mathbf{P}(\mathbf{A}[u, v] = 1) \propto \mathbf{S}[u, v]. \tag{1.7}$$

Thus, in order to approach the relation prediction task using a neighborhood overlap measure, one simply needs to set a threshold to determine when to predict the existence of an edge. Note that in the relation prediction setting we generally assume that we only know a subset of the true edges $E_{\text{train}} \subset E$. Our hope is that *node-node similarity measures* computed on the training edges will lead to accurate predictions about the existence of test (i.e., unseen) edges.

node-node similarity measures

1.2.3 Towards Learned Representations

In the previous sections, we saw a number of traditional approaches to learning over graphs. However, the approaches discussed in this section — and especially the node and graph-level statistics — are limited due to the fact that they require careful, hand-engineered statistics and measures. The following chapters in this book introduce alternative approach to learning over graphs: *graph representation learning*. Instead of extracting hand-engineered features, we will seek to *learn* representations that encode structural information about the graph.

1.3 Neighborhood Reconstruction Methods

node embeddings

This part of the book is concerned with methods for learning *node embeddings*. The goal of these methods is to encode nodes as low-dimensional vectors that summarize their graph position and the structure of their local graph neighborhood. In other words, we want to project nodes into a latent space, where geometric relations in this latent space correspond to relationships (e.g., edges) in the original graph or network (Figure 1.1).

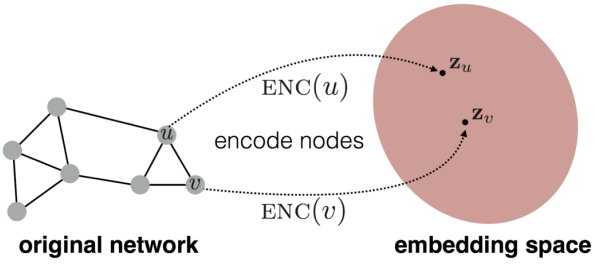

Figure 1.1: Illustration of the node embedding problem. Our goal is to learn an encoder (enc), which maps nodes to a low-dimensional embedding space. These embeddings are optimized so that distances in the embedding space reflect the relative positions of the nodes in the original graph.

1.3.1 An Encoder-Decoder Perspective

encoding and decoding graphs

We organize our discussion of node embeddings based upon the framework of *encoding and decoding graphs*. This way of viewing graph representation learning will reoccur throughout the book, and our presentation of node embedding methods based on this perspective closely follows [7].

In the encoder-decoder framework, we view the graph representation learning problem as involving two key operations. First, an *encoder* model maps each node in the graph into a low-dimensional vector or embedding. Next, a *decoder*

model takes the low-dimensional node embeddings and uses them to reconstruct information about each node's neighborhood in the original graph. This idea is summarized in Figure 1.2.

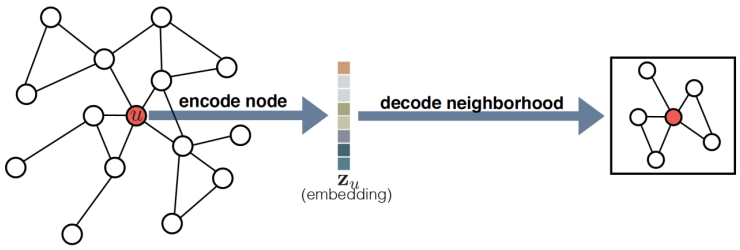

Figure 1.2: Overview of the encoder-decoder approach. The encoder maps the node u to a low-dimensional embedding $\mathbf{z}_u$. The decoder then uses $\mathbf{z}_u$ to reconstruct u's local neighborhood information.

The Encoder

encoder

Formally, the *encoder* is a function that maps nodes $v \in V$ to vector embeddings $\mathbf{z}_v \in \mathbb{R}^d$ (where $\mathbf{z}_v$ corresponds to the embedding for node $v \in V$). In the simplest case, the encoder has the following signature:

$$\text{ENC} : V \to \mathbb{R}^d, \tag{1.8}$$

meaning that the encoder takes node IDs as input to generate the node embeddings. In most work on node embeddings, the encoder relies on what we call the *shallow embedding* approach, where this encoder function is simply an embedding lookup based on the node ID. In other words, we have that

shallow embedding

$$\text{ENC}(v) = \mathbf{Z}[v], \tag{1.9}$$

where $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$ is a matrix containing the embedding vectors for all nodes and $\mathbf{Z}[v]$ denotes the row of $\mathbf{Z}$ corresponding to node v.

The Decoder

decoder

The role of the *decoder* is to reconstruct certain graph statistics from the node embeddings that are generated by the encoder. For example, given a node embedding $\mathbf{z}_u$ of a node u, the decoder might attempt to predict u's set of neighbors $\mathcal{N}(u)$ or its row $\mathbf{A}[u]$ in the graph adjacency matrix.

While many decoders are possible, the standard practice is to define *pairwise decoders*, which have the following signature:

pairwise decoders

$$\text{DEC} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^+. \tag{1.10}$$

Pairwise decoders can be interpreted as predicting the relationship or similarity between pairs of nodes. For instance, a simple pairwise decoder could predict whether two nodes are neighbors in the graph.

Applying the pairwise decoder to a pair of embeddings $(\mathbf{z}_u, \mathbf{z}_v)$ results in the *reconstruction* of the relationship between nodes u and v. The goal is optimize the encoder and decoder to minimize the reconstruction loss so that

$$\text{DEC}(\text{ENC}(u), \text{ENC}(v)) = \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \approx \mathbf{S}[u, v]. \tag{1.11}$$

Here, we assume that $\mathbf{S}[u, v]$ is a graph-based similarity measure between nodes. For example, the simple reconstruction objective of predicting whether two nodes are neighbors would correspond to $\mathbf{S}[u, v] := \mathbf{A}[u, v]$. However, one can define $\mathbf{S}[u, v]$ in more general ways as well, for example, by leveraging any of the pairwise neighborhood overlap statistics discussed in subsection 1.2.2.

Optimizing an Encoder-Decoder Model

To achieve the reconstruction objective (Equation 1.11), the standard practice is to minimize an empirical reconstruction loss $\mathcal{L}$ over a set of training node pairs $\mathcal{D}$:

$$\mathcal{L} = \sum_{(u,v) \in \mathcal{D}} \ell(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v), \mathbf{S}[u, v]), \tag{1.12}$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a loss function measuring the discrepancy between the decoded (i.e., estimated) similarity values $\text{DEC}(\mathbf{z}_u, \mathbf{z}_v)$ and the true values $\mathbf{S}[u, v]$. Depending on the definition of the decoder (DEC) and similarity function ($\mathbf{S}$), the loss function ℓ might be a mean-squared error or even a classification loss, such as cross entropy. Thus, the overall objective is to train the encoder and the decoder so that pairwise node relationships can be effectively reconstructed on the training set $\mathcal{D}$. Most approaches minimize the loss in Equation 1.12 using stochastic gradient descent, but there are certain instances when more specialized optimization methods (e.g., based on matrix factorization) can be used.

Overview of the Encoder-Decoder Approach

Table 1.1 applies this encoder-decoder perspective to summarize several well-known node embedding methods|all of which use the shallow encoding approach. The key benefit of the encoder-decoder framework is that it allows one to succinctly define and compare different embedding methods based on (i) their decoder function, (ii) their graph-based similarity measure, and (iii) their loss function.

Method	Decoder $\text{DEC}(\mathbf{z}_u, \mathbf{z}_v)$	Similarity measure $\mathbf{S}[u, v]$	Loss function $\mathcal{L}$
Lap. Eigenmaps	$\|\mathbf{z}_u - \mathbf{z}_v\|_2^2$	general	$\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]$
Graph Fact.	$\mathbf{z}_u^T \mathbf{z}_v$	$\mathbf{A}[u, v]$	$\|\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]\|_2^2$
GraRep	$\mathbf{z}_u^T \mathbf{z}_v$	$\mathbf{A}[u, v], \ldots, \mathbf{A}^k[u, v]$	$\|\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]\|_2^2$
HOPE	$\mathbf{z}_u^T \mathbf{z}_v$	general	$\|\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]\|_2^2$
DeepWalk	$\frac{e^{\mathbf{z}_u^T \mathbf{z}_v}}{\sum_{k \in V} e^{\mathbf{z}_u^T \mathbf{z}_k}}$	$p_G(v\|u)$	$-\mathbf{S}[u, v] \log(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v))$
node2vec	$\frac{e^{\mathbf{z}_u^T \mathbf{z}_v}}{\sum_{k \in V} e^{\mathbf{z}_u^T \mathbf{z}_k}}$	$p_G(v\|u)$ (biased)	$-\mathbf{S}[u, v] \log(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v))$

Table 1.1: A summary of some well-known shallow embedding algorithms. Note that the decoders and similarity functions for the random-walk based methods are asymmetric, with the similarity function $p_G(v|u)$ corresponding to the probability of visiting v on a fixed-length random walk starting from u.

1.3.2 Limitations of Shallow Embeddings

This focus of this chapter has been on shallow embedding methods. In these approaches, the encoder model that maps nodes to embeddings is simply an embedding lookup (Equation 1.9), which trains a unique embedding for each node in the graph. However, it is also important to note that shallow embedding approaches suffer from some important drawbacks:

1. The first issue is that they do not share any parameters between nodes in the encoder, since the encoder directly optimizes a unique embedding vector for each node. This lack of parameter sharing is both statistically and computationally inefficient.

2. A second key issue is that they do not leverage node features in the encoder.

3. Lastly — and perhaps most importantly — they are inherently *transductive*. These methods can only generate embeddings for nodes that were present during the training phase. Generating embeddings for new nodes — which are observed after the training phase — is not possible unless additional optimizations are performed to learn the embeddings for these nodes. This restriction prevents shallow embedding methods from being used on *inductive* applications, which involve generalizing to unseen nodes after training.

To alleviate these limitations, shallow encoders can be replaced with more sophisticated encoders that depend more generally on the structure and attributes of the graph. We will discuss the most popular paradigm to define such encoders, i.e.., graph neural networks (GNNs).

1.4 The Graph Neural Network Model

The first part of this book discussed approaches for learning low-dimensional embeddings of the nodes in a graph. The node embedding approaches we discussed used a shallow embedding approach to generate representations of nodes, where we simply optimized a unique embedding vector for each node. In this chapter, we turn our focus to more complex encoder models. We will introduce the graph neural network (GNN) formalism, which is a general framework for defining deep neural networks on graph data. The key idea is that we want to generate representations of nodes that actually depend on the structure of the graph, as well as any feature information we might have.

The primary challenge in developing complex encoders for graph-structured data is that our usual deep learning toolbox does not apply. For example, convolutional neural networks (CNNs) are well-defined only over grid-structured inputs (e.g., images), while recurrent neural networks (RNNs) are well-defined only over sequences (e.g., text). To define a deep neural network over general graphs, we need to define a new kind of deep learning architecture.

Permutation invariance and equivariance One reasonable idea for defining a deep neural network over graphs would be to simply use the adjacency matrix as input to a deep neural network. For example, to generate an embedding of an entire graph we could simply flatten the adjacency matrix and feed the result to a multi-layer perceptron (MLP):

$$\mathbf{z}_G = \text{MLP}(A[1] \oplus A[2] \oplus \ldots \oplus A[|\mathcal{V}|]); \tag{1.13}$$

where $A[i] \in \mathbf{R}^{|\mathcal{V}|}$ denotes a row of the adjacency matrix and we use $\oplus$ to denote vector concatenation.

The issue with this approach is that it *depends on the arbitrary ordering of nodes that we used in the adjacency matrix*. In other words, such a model is not *permutation invariant*, and a key desideratum for designing neural networks over graphs is that they should permutation invariant (or equivariant). In mathematical terms, any function f that takes an adjacency matrix A as input should ideally satisfy one of the two following properties:

$$f(\mathbf{P}\mathbf{A}\mathbf{P}^T) = f(A) \qquad \text{(Permutation Invariance)} \tag{1.14}$$

$$f(\mathbf{P}\mathbf{A}\mathbf{P}^T) = \mathbf{P}f(A) \qquad \text{(Permutation Equivariance)}, \tag{1.15}$$

permutation invariance

permutation equivariance

where $\mathbf{P}$ is a permutation matrix. *Permutation invariance* means that the function does not depend on the arbitrary ordering of the rows/columns in the adjacency matrix, while *permutation equivariance* means that the output of f is permuted in an consistent way when we permute the adjacency matrix. (The shallow encoders are an example of permutation equivariant functions.) Ensuring invariance or equivariance is a key challenge when we are learning over graphs, and we will revisit issues surrounding permutation equivariance and invariance often in the ensuing chapters.

1.4.1 Neural Message Passing

The defining feature of a GNN is that it uses a form of neural message passing in which vector messages are exchanged between nodes and updated using neural networks.

In the rest of this chapter, we will detail the foundations of this neural message passing framework. We will focus on the message passing framework itself and defer discussions of training and optimizing GNN models to the next Section. The bulk of this section will describe how we can take an input graph $G = (V, E)$, along with a set of node features $\mathbf{X} \in \mathbb{R}^{d \times |V|}$, and use this information to generate node embeddings $\mathbf{z}_u, \forall u \in V$. However, we will also discuss how the GNN framework can be used to generate embeddings for subgraphs and entire graphs.

During each message-passing iteration in a GNN, a *hidden embedding* $\mathbf{h}_u^{(k)}$ corresponding to each node $u \in \mathcal{V}$ is updated according to information aggregated from u's graph neighborhood $\mathcal{N}(u)$. This message-passing update can be expressed as follows:

hidden embedding

$$\mathbf{h}_u^{(k+1)} = \textsc{update}^{(k)}\left(\mathbf{h}_u^{(k)}; \textsc{aggregate}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \tag{1.16}$$

$$= \textsc{update}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right), \tag{1.17}$$

where UPDATE and AGGREGATE are arbitrary differentiable functions (i.e., neural networks) and

$$\mathbf{m}_{\mathcal{N}(u)} = \textsc{aggregate}(\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\}) \tag{1.18}$$

is the "message" that is aggregated from u's graph neighborhood $\mathcal{N}(u)$. We use superscripts to distinguish the embeddings and functions at different iterations of message passing. The different iterations of message passing are also sometimes known as the different "layers" of the GNN.

At each iteration k of the GNN, the AGGREGATE function takes as input the set of embeddings of the nodes in u's graph neighborhood $\mathcal{N}(u)$ and generates a message $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \textsc{aggregate}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$ based on this aggregated neighborhood information (*neighborhood aggregation operation*). The update function UPDATE then combines the message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ with the previous embedding $\mathbf{h}_u^{(k1)}$ of node u to generate the updated embedding $\mathbf{h}_u^{(k)}$. The initial embeddings at $k = 0$ are set to the input features for all the nodes, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in \mathcal{V}$. After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings for each node, i.e.,

neighborhood aggregation operation

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}. \tag{1.19}$$

Note that since the AGGREGATE function takes a set as input, GNNs defined in this way are permutation equivariant by design.

In cases where no node features are available, there are still several options. One option is to use node statistics to define features. Another popular approach is to use identity features, where we associate each node with a one-hot indicator feature, which uniquely identifies that node. Note, however, that the using identity features makes the model transductive and incapable of generalizing to unseen nodes.

The basic intuition behind the GNN message-passing framework is straightforward: at each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph.

structural information

But what kind of "information" do these node embeddings actually encode? Generally, this information comes in two forms. On the one hand there is *structural information* about the graph. For example, after k iterations of GNN message passing, the embedding $\mathbf{h}_u^{(k)}$ of node u might encode information about the degrees of all the nodes in u's k-hop neighborhood.

feature-based

In addition to structural information, the other key kind of information captured by GNN node embedding is *feature-based*. After k iterations of GNN message passing, the embeddings for each node also encode information about all the features in their k-hop neighborhood. This local feature-aggregation behaviour of GNNs is analogous to the behavior of the convolutional kernels in convolutional neural networks (CNNs). However, whereas CNNs aggregate feature information from spatially-defined patches in an image, GNNs aggregate information based on local graph neighborhoods.

The Basic GNN

So far, we have discussed the GNN framework in a relatively abstract fashion as a series of message-passing iterations using UPDATE and AGGREGATE functions (Equation 1.16). In order to translate the abstract GNN framework defined in Equation 1.16 into something we can implement, we must give concrete instantiations to these UPDATE and AGGREGATE functions. We begin here with the most basic GNN framework. The basic GNN message passing is defined as

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v\in\mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right), \tag{1.20}$$

where $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d^{(k)}\times d^{(k-1)}}$ are trainable parameter matrices and σ denotes an elementwise non-linearity (e.g., a tanh or ReLU). The bias term $b^{(k)} \in \mathbb{R}^{d^{(k)}}$ is often omitted for notational simplicity, but including the bias term can be important to achieve strong performance. In this equation — and throughout the remainder of the review — we use superscripts to differentiate parameters, embeddings, and dimensionalities in different layers of the GNN.

The message passing in the basic GNN framework is analogous to a standard multi-layer perceptron (MLP) or Elman-style recurrent neural network, i.e.,

Elman RNN, as it relies on linear operations followed by a single elementwise non-linearity. We first sum the messages incoming from the neighbors; then, we combine the neighborhood information with the node's previous embedding using a linear combination; and finally, we apply an elementwise non-linearity.

We can equivalently define the basic GNN through the UPDATE and AGGREGATE functions:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v, \tag{1.21}$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)}\right), \tag{1.22}$$

where we recall that we use

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)},\ \forall v \in \mathcal{N}(u)\}) \tag{1.23}$$

as a shorthand to denote the message that has been aggregated from u's graph neighborhood. Note also that we omitted the superscript denoting the iteration in the above equations, which we will often do for notational brevity. In general, the parameters $\mathbf{W}_{\text{self}}, \mathbf{W}_{\text{neigh}}$ and $\mathbf{b}$ can be shared across the GNN message passing iterations or trained separately for each layer.

Node vs. graph-level equations. In the description of the basic GNN model above, we defined the core message-passing operations at the node level. However, it is important to note that many GNNs can also be succinctly defined using graph-level equations. In the case of a basic GNN, we can write the graph-level definition of the model as follows:

$$\mathbf{H}^{(k)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}_{\text{neigh}}^{(k)} + \mathbf{H}^{(k-1)}\mathbf{W}_{\text{self}}^{(k)}\right), \tag{1.24}$$

where $\mathbf{H}^{(k)} \in \mathbb{R}^{|V| \times d}$ denotes the matrix of node representations at layer k in the GNN (with each node corresponding to a row in the matrix), $\mathbf{A}$ is the graph adjacency matrix, and we have omitted the bias term for notational simplicity. While this graph-level representation is not easily applicable to all GNN models — such as the attention-based models — it is often more succinct and also highlights how many GNNs can be efficiently implemented using a small number of sparse matrix operations.

Message Passing with Self-loops

As a simplification of the neural message passing approach, it is common to add self-loops to the input graph and omit the explicit update step. In this approach we define the message passing simply as

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)},\ \forall v \in \mathcal{N}(u) \cup \{u\}\}), \tag{1.25}$$

where now the aggregation is taken over the set $\mathcal{N}(u) \cup \{u\}$, i.e., the node's neighbors as well as the node itself. The benefit of this approach is that we no

longer need to define an explicit update function, as the update is implicitly defined through the aggregation method.

In the case of the basic GNN, adding self-loops is equivalent to sharing parameters between the $\mathbf{W}_{\text{self}}$ and $\mathbf{W}_{\text{neigh}}$ matrices, which gives the following graph-level update:

$$\mathbf{H}^{(k)} = \sigma\left((\mathbf{A} + I)\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right). \tag{1.26}$$

self-loop approach

In the following chapters we will refer to this as the *self-loop approach*.

1.4.2 Graph convolutional networks (GCNs)

The basic GNN can be improved upon and generalized in many ways: both the AGGREGATE operator and the UPDATE can be generalized and improved upon.

The most basic neighborhood aggregation operation (Equation 1.21) simply takes the sum of the neighbor embeddings. One issue with this approach is that it can be unstable and highly sensitive to node degrees. One solution to this problem is to simply *normalize the aggregation operation* based upon the degrees of the nodes involved. The simplest approach is to just take an average rather than sum:

$$\mathbf{m}_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v}{|\mathcal{N}(u)|}, \tag{1.27}$$

but researchers have also found success with other normalization factors, such as the following *symmetric normalization*:

symmetric normalization

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}. \tag{1.28}$$

graph convolutional network (GCN)

One of the most popular baseline graph neural network models — the *graph convolutional network (GCN)* — employs the symmetric-normalized aggregation as well as the self-loop approach. The GCN model thus defines the message passing function as

$$h_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{1.29}$$

To normalize or not to normalize? Proper normalization can be essential to achieve stable and strong performance when using a GNN. It is important to note, however, that normalization can also lead to a loss of information. For example, after normalization, it can be hard (or even impossible) to use the learned embeddings to distinguish between nodes of different degrees, and various other structural graph features can be obscured by normalization. The use of normalization is thus an application-specific question. Usually, normalization

is most helpful in tasks where node feature information is far more useful than structural information, or where there is a very wide range of node degrees that can lead to instabilities during optimization.

Set Aggregators

Neighborhood normalization can be a useful tool to improve GNN performance, but can we do more to improve the AGGREGATE operator? Is there perhaps something more sophisticated than just summing over the neighbor embeddings?

The *neighborhood aggregation operation* is fundamentally a set function. We are given a set of neighbor embeddings $\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\}$ and must map this set to a single vector $\mathbf{m}_{\mathcal{N}(u)}$. The fact that $\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\}$ is a set is in fact quite important: there is no natural ordering of a nodes' neighbors, and any aggregation function we define must thus be permutation invariant.

One principled approach to define an aggregation function is based on the theory of permutation invariant neural networks. An aggregation function with the following form is a *universal set function approximator*:

universal set function approximator

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{MLP}_\theta \left(\sum_{v \in \mathcal{N}(u)} \mathbf{MLP}_\phi(\mathbf{h}_v) \right), \tag{1.30}$$

where as usual we use $\mathbf{MLP}_\theta$ to denote an arbitrarily deep multi-layer perceptron parameterized by some trainable parameters θ. In other words, theoretical results show that any permutation-invariant function that maps a set of embeddings to a single embedding can be approximated to an arbitrary accuracy by a model following Equation 1.30. Set pooling approaches based on Equation 1.30 often lead to small increases in performance, though they also introduce an increased risk of overfitting, depending on the depth of the MLPs used.

Janossy pooling employs a different approach entirely: instead of using a permutationinvariant reduction (e.g., a sum or mean), we apply a *permutation-sensitive function* and average the result over many possible permutations.

Janossy pooling

Neighborhood Attention

In addition to more general forms of set aggregation, a popular strategy for improving the aggregation layer in GNNs is to apply attention. The basic idea is to assign an attention weight or importance to each neighbor, which is used to weigh this neighbor's influence during the aggregation step. The first GNN model to apply this style of attention was the *Graph Attention Network (GAT)*, which uses attention weights to define a weighted sum of the neighbors:

Graph Attention Network (GAT)

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v, \tag{1.31}$$

where $\alpha_{u,v}$ denotes the attention on neighbor $v \in \mathcal{N}(u)$ when we are aggregating information at node u. In the original GAT paper, the attention weights are defined as

$$\alpha_{u,v} = \frac{\exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v]\right)}{\sum_{v' \in \mathcal{N}(u)} \exp\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}]\right)}, \tag{1.32}$$

where $\mathbf{a}$ is a trainable attention vector, $\mathbf{W}$ is a trainable matrix, and $\oplus$ denotes the concatenation operation. Adding attention is a useful strategy for increasing the representational capacity of a GNN model, especially in cases where you have prior knowledge to indicate that some neighbors might be more informative than others.

In many ways the UPDATE operator plays an equally important role as the AGGREGATE operator in defining the power and inductive bias of the GNN model. So far, we have seen the basic GNN approach — where the update operation involves a linear combination of the node's current embedding with the message from its neighbors — as well as the self-loop approach, which simply involves adding a self-loop to the graph before performing neighborhood aggregation. In this section, we turn our attention to more diverse generalizations of the UPDATE operator.

over-smoothing

One common issue with GNNs — which generalized update methods can help to address — is known as *over-smoothing*. The essential idea of over-smoothing is that after several iterations of GNN message passing, the representations for all the nodes in the graph can become very similar to one another. This tendency is especially common in basic GNN models and models that employ the selfloop update approach. Over-smoothing is problematic because it makes it impossible to build deeper GNN models — which leverage longer-term dependencies in the graph — since these deep GNN models tend to just generate over-smoothed embeddings.

A theorem states that when we are using a K-layer GCN-style model, the influence of node u and node v is proportional the probability of reaching node v on a K-step random walk starting from node u. An important consequence of this, however, is that as $K \to \infty$ the influence of every node approaches the stationary distribution of random walks over the graph, meaning that local neighborhood information is lost. Moreover, in many real-world graphs — which contain high-degree nodes and resemble so-called "expander" graphs — it only takes $k = O(\log(|V|)$ steps for the random walk starting from any node to converge to an almost-uniform distribution. Thus, when using simple GNN models — and especially those with the self-loop update approach — building deeper models can actually hurt performance. As more layers are added we lose information about local neighborhood structures and our learned embeddings become over-smoothed, approaching an almost-uniform distribution.

Concatenation and Skip-Connections

As discussed above, over-smoothing is a core issue in GNNs. Over-smoothing occurs when node-specific information becomes "washed out" or "lost" after several iterations of GNN message passing. Intuitively, we can expect oversmoothing in cases where the information being aggregated from the node neighbors during message passing begins to dominate the updated node representations. A natural way to alleviate this issue is to use vector concatenations or skip connections, which try to directly preserve information from previous rounds of message passing during the update step.

One of the simplest skip connection updates employs a concatenation to preserve more node-level information during message passing:

$$\textsc{update}_{\text{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\textsc{update}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u], \tag{1.33}$$

where we simply concatenate the output of the base update function with the node's previous-layer representation. Again, the key intuition here is that we encourage the model to disentangle information during message passing — separating the information coming from the neighbors (i.e., $\mathbf{m}_{\mathcal{N}(u)}$) from the current representation of each node (i.e., $\mathbf{h}_u$).

However, in addition to concatenation, we can also employ other forms of skip-connections, such as the *linear interpolation method*

linear interpolation method

$$\textsc{update}_{\text{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \alpha_1 \circ \textsc{update}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \alpha_2 \mathbf{h}_u, \tag{1.34}$$

where $\alpha_1, \alpha_2 \in [0,1]^d$ are gating vectors with $\alpha_2 = 1 - \alpha_1$ and $\circ$ denotes elementwise multiplication. In this approach, the final updated representation is a linear interpolation between the previous representation and the representation that was updated based on the neighborhood information. The gating parameters α_1 can be learned jointly with the model in a variety of ways.

In general, these concatenation and residual connections are simple strategies that can help to alleviate the over-smoothing issue in GNNs, while also improving the numerical stability of optimization. Indeed, similar to the utility of residual connections in convolutional neural networks (CNNs), applying these approaches to GNNs can facilitate the training of much deeper models. In practice these techniques tend to be most useful for node classification tasks with moderately deep GNNs (e.g., 2-5 layers), and they excel on tasks that exhibit homophily, i.e., where the prediction for each node is strongly related to the features of its local neighborhood.

Gated Updates

One way to view the GNN message passing algorithm is that the aggregation function is receiving an observation from the neighbors, which is then used to update the hidden state of each node. In this view, we can directly apply methods used to update the hidden state of RNN architectures based on observations.

For instance, one of the earliest GNN architectures defines the update function as

$$\mathbf{h}_u^{(k)} = \text{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}), \tag{1.35}$$

where GRU denotes the update equation of the gated recurrent unit (GRU) cell. Other approaches have employed updates based on the LSTM architecture.

In general, any update function defined for RNNs can be employed in the context of GNNs. We simply replace the hidden state argument of the RNN update function (usually denoted $\mathbf{h}^{(t)}$) with the node's hidden state, and we replace the observation vector (usually denoted $\mathbf{x}^{(t)}$) with the message aggregated from the local neighborhood. Importantly, the parameters of this RNN-style update are always shared across nodes (i.e., we use the same LSTM or GRU cell to update each node). In practice, researchers usually share the parameters of the update function across the message-passing layers of the GNN as well.

These gated updates are very effective at facilitating deep GNN architectures (e.g., more than 10 layers) and preventing over-smoothing. Generally, they are most useful for GNN applications where the prediction task requires complex reasoning over the global structure of the graph, such as applications for program analysis or combinatorial optimization [Selsam et al., 2019].

1.4.3 Graph Pooling

The neural message passing approach produces a set of node embeddings, but what if we want to make predictions at the graph level? In other words, we have been assuming that the goal is to learn node representations $\mathbf{z}_u$, $\forall u \in V$, but what if we want to learn an embedding $\mathbf{z}_G$ for the entire graph G? This task is often referred to as *graph pooling*, since our goal is to pool together the node embeddings in order to learn an embedding of the entire graph.

graph pooling

Similar to the AGGREGATE operator, the task of graph pooling can be viewed as a problem of learning over sets. We want to design a pooling function f_p, which maps a set of node embeddings $\{\mathbf{z}_1, \ldots, \mathbf{z}_{|V|}\}$ to an embedding $\mathbf{z}_G$ that represents the full graph.

There are two approaches that are most commonly applied for learning graph-level embeddings via set pooling. The first approach is simply to take a sum (or mean) of the node embeddings:

$$\mathbf{z}_G = \frac{\sum_{v \in V} \mathbf{z}_u}{f_n(|V|)}, \tag{1.36}$$

where f_n is some normalizing function (e.g., the identity function, so $\mathbf{z}_G = (\sum_{v \in V} \mathbf{z}_u)/|V|$). While quite simple, pooling based on the sum or mean of the node embeddings is often sufficient for applications involving small graphs. The second popular set-based approach uses a combination of LSTMs and attention to pool the node embeddings (which we won't see).

One limitation of the set pooling approaches is that they do not exploit the structure of the graph. While it is reasonable to consider the task of graph pooling as simply a set learning problem, there can also be benefits from exploiting the graph topology at the pooling stage. One popular strategy to accomplish this is to perform *graph clustering or coarsening* as a means to pool the node representations.

1.4.4 Generalized Message Passing

The presentation in this chapter so far has focused on the most popular style of GNN message passing, which operates largely at the node level. However, the GNN message passing approach can also be generalized to leverage edge and graph-level information at each stage of message passing. For example, in the more general approach proposed by [2], we define each iteration of message passing according to the following equations:

$$\mathbf{h}_{(u,v)}^{(k)} = \text{UPDATE}_{\text{edge}}(\mathbf{h}_{(u,v)}^{(k-1)}, \mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}, \mathbf{h}_G^{(k-1)}) \tag{1.37}$$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}_{\text{node}}(\{\mathbf{h}_{(u,v)}^{(k)},\ \forall v \in \mathcal{N}(u)\}) \tag{1.38}$$

$$\mathbf{h}_u^{(k)} = \text{UPDATE}_{\text{node}}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}, \mathbf{h}_G^{(k-1)}) \tag{1.39}$$

$$\mathbf{h}_G^{(k)} = \text{UPDATE}_{\text{graph}}(\mathbf{h}_G^{(k-1)}, \{\mathbf{h}_u^{(k)}, \forall u \in V\}, \{\mathbf{h}_{(u,v)}^{(k)}, \forall (u,v) \in E\}). \tag{1.40}$$

The important innovation in this generalized message passing framework is that, during message passing, we generate hidden embeddings $\mathbf{h}_{(u,v)}^{(k)}$ for each edge in the graph, as well as an embedding $\mathbf{h}_G^{(k)}$ corresponding to the entire graph. This allows the message passing model to easily integrate edge and graph-level features, and recent work has also shown this generalized message passing approach to have benefits compared to a standard GNN in terms of logical expressiveness. Generating embeddings for edges and the entire graph during message passing also makes it trivial to define loss functions based on graph or edge-level classification tasks.

In terms of the message-passing operations in this generalized message-passing framework, we first update the edge embeddings based on the embeddings of their incident nodes (Equation 1.37). Next, we update the node embeddings by aggregating the edge embeddings for all their incident edges (Equation 1.38 and 1.39). The graph embedding is used in the update equation for both node and edge representations, and the graph-level embedding itself is updated by aggregating over all the node and edge embeddings at the end of each iteration (Equation 1.40). All of the individual update and aggregation operations in such a generalized message-passing framework can be implemented using the techniques discussed earlier (e.g., using a pooling method to compute the graph-level update).

1.5 Graph Neural Networks in Practice

In section 1.1, we introduced a number of graph neural network (GNN) architectures. However, we did not discuss how these architectures are optimized and what kinds of loss functions and regularization are generally used. In this section, we will turn our attention to some of these practical aspects of GNNs. We will discuss some representative applications and how GNNs are generally optimized in practice, including a discussion of unsupervised pre-training methods that can be particularly effective. We will also introduce common techniques used to regularize and improve the efficiency of GNNs.

1.5.1 Applications and Loss Functions

In the vast majority of current applications, GNNs are used for one of three tasks: node classification, graph classification, or relation prediction. As discussed in section 1.4, these tasks reflect a large number of real-world applications, such as predicting whether a user is a bot in a social network (node classification), property prediction based on molecular graph structures (graph classification), and content recommendation in online platforms (relation prediction). In this section, we briefly describe how these tasks translate into concrete loss functions for GNNs, and we also discuss how GNNs can be pre-trained in an unsupervised manner to improve performance on these downstream tasks.

GNNs for Node Classification

node classification

The standard way to apply GNNs to such a *node classification* task is to train GNNs in a fully-supervised manner, where we define the loss using a softmax classification function and negative log-likelihood loss:

$$\mathcal{L} = \sum_{u \in V_{\text{train}}} -\log(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)), \tag{1.41}$$

Here, we assume that $\mathbf{y}_u \in \mathbb{Z}^c$ is a one-hot vector indicating the class of training node $u \in V_{\text{train}}$ and we use $\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)$ to denote the predicted probability that the node belongs to the class $\mathbf{y}_u$, computed via the softmax function:

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^\top \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_u^\top \mathbf{w}_j}}, \tag{1.42}$$

where $\mathbf{w}_i \in \mathbb{R}^d, i = 1, \ldots, c$ are trainable parameters.

Supervised, semi-supervised, transductive, and inductive Note that — as discussed in section 1.1 — the node classification setting is often referred to both as supervised and semi-supervised. One important factor when applying these terms is whether and how different nodes are used during training the GNN. Generally, we can distinguish between three types of nodes:

1. There is the set of training nodes, V_{train}. These nodes are included in the GNN message passing operations, and they are also used to compute the loss, e.g., via Equation 1.41.

2. In addition to the training nodes, we can also have transductive test nodes, V_{trans}. These nodes are unlabeled and not used in the loss computation, but these nodes — and their incident edges — are still involved in the GNN message passing operations. In other words, the GNN will generate hidden representations $\mathbf{h}_u^{(k)}$ for the nodes in $u \in V_{\text{trans}}$ during the GNN message passing operations. However, the final layer embeddings $\mathbf{z}_u$ for these nodes will not be used in the loss function computation.

3. Finally, we will also have inductive test nodes, V_{ind}. These nodes are not used in either the loss computation or the GNN message passing operations during training, meaning that these nodes — and all of their edges — are completely unobserved while the GNN is trained.

The term semi-supervised is applicable in cases where the GNN is tested on transductive test nodes, since in this case the GNN observes the test nodes (but not their labels) during training. The term inductive node classification is used to distinguish the setting where the test nodes|and all their incident edges|are completely unobserved during training. An example of inductive node classification would be training a GNN on one subgraph of a citation network and then testing it on a completely disjoint subgraph.

GNNs for Graph Classification

graph classification

In *graph classification*, a softmax classification loss — analogous to Equation 1.41 — is often used, with the key difference that the loss is computed with graph-level embeddings $\mathbf{z}_{G_i}$ over a set of labeled training graphs $\mathcal{G} = \{G_1, \ldots, G_n\}$. In recent years, GNNs have also witnessed success in regression tasks involving graph data. In these instances, it is standard to employ a squared-error loss of the following form:

$$\mathcal{L} = \sum_{G_i \in \mathcal{G}} \|\mathbf{MLP}(\mathbf{z}_{G_i}) - y_{G_i}\|_2^2, \tag{1.43}$$

where **MLP** is a densely connected neural network with a univariate output and $y_{G_i} \in \mathbb{R}$ is the target value for training graph G_i.

GNNs for Edge Prediction

While classification tasks are by far the most popular application of GNNs, GNNs are also used in in edge prediction tasks, such as recommender systems and knowledge graph completion. In these applications, the standard practice is to employ the pairwise node embedding loss functions. In principle, GNNs can be combined with any of the pairwise loss functions discussed previously, with the output of the GNNs replacing the shallow embeddings.

Chapter 2

Graph Representation Learning - Video

Notes of the video [5]. Related video: [10].

Graphs are a general and universal language for describing and modelling complex systems/data.

In a graph setting, we are not considering a set of independent points, but really the whole object that we're trying to actually do learning upon is bound up in the interconnections or the relationships between these points. So rather than a set of individual data points we're considering the relationships between them.

Even trying to tell whether or not two graphs are the same is NP-indeterminate (it's believed not to be solved in polynomial time), since there is no standard or canonical way to order the nodes in the adjacency matrices (in a graph there is no up and down like in an image!).

Chapter 3

Relational inductive biases, deep learning, and graph networks

Summary of [2].

3.1 Introduction

A key signature of human intelligence is the ability to make "infinite use of finite means", in which a small set of elements (such as words) can be productively composed in limitless ways (such as into new sentences). This reflects the principle of *combinatorial generalization*, that is, constructing new inferences, predictions, and behaviors from known building blocks. Here we explore how to improve modern AI's capacity for combinatorial generalization by biasing learning towards structured representations and computations, and in particular, systems that operate on graphs. The question of how to build artificial systems which exhibit combinatorial generalization has been at the heart of AI since its origins, and was central to many structured approaches.

combinatorial generalization

Recently, a class of models has arisen at the intersection of deep learning and structured approaches, which focuses on approaches for reasoning about explicitly structured data, in particular graphs. What these approaches all have in common is a capacity for performing computation over discrete entities and the relations between them. What sets them apart from classical approaches is how the representations and structure of the entities and relations — and the corresponding computations — can be learned, relieving the burden of needing to specify them in advance. Crucially, these methods carry strong *relational inductive biases*, in the form of specific architectural assumptions, which guide

these approaches towards learning about entities and relations, which we, joining many others, suggest are an essential ingredient for human-like intelligence.

3.2 Relational inductive biases

We define *structure* as the product of composing a set of known building blocks. "Structured representations" capture this composition (i.e., the arrangement of the elements) and "structured computations" operate over the elements and their composition as a whole. *Relational reasoning*, then, involves manipulating structured representations of *entities* and *relations*, using *rules* for how they can be composed. We use these terms to capture notions from cognitive science, theoretical computer science, and AI, as follows:

relational reasoning

entity

relation

rule

- An *entity* is an element with attributes, such as a physical object with a size and mass.
- A *relation* is a property between entities. Relations between two objects might include same size as, heavier than, and distance from. Relations can have attributes as well. The relation more than X times heavier than takes an attribute, X, which determines the relative weight threshold for the relation to be true vs. false. Relations can also be sensitive to the global context. For a stone and a feather, the relation falls with greater acceleration than depends on whether the context is in air vs. in a vacuum. Here we focus on pairwise relations between entities.
- A *rule* is a function (like a non-binary logical predicate) that maps entities and relations to other entities and relations, such as a scale comparison like is entity X large? and is entity X heavier than entity Y?. Here we consider rules which take one or two arguments (unary and binary), and return a unary property value.

inductive bias

An *inductive bias* allows a learning algorithm to prioritize one solution (or interpretation) over another, independent of the observed data. In a Bayesian model, inductive biases are typically expressed through the choice and parameterization of the prior distribution. In other contexts, an inductive bias might be a regularization term added to avoid overfitting, or it might be encoded in the architecture of the algorithm itself. Inductive biases often trade flexibility for improved sample complexity and can be understood in terms of the bias-variance tradeoff. Ideally, inductive biases both improve the search for solutions without substantially diminishing performance, as well as help find solutions which generalize in a desirable way; however, mismatched inductive biases can also lead to suboptimal performance by introducing constraints that are too strong.

relational inductive bias

Many approaches in machine learning and AI which have a capacity for relational reasoning use a *relational inductive bias*. While not a precise, formal

definition, we use this term to refer generally to inductive biases which impose constraints on relationships and interactions among entities in a learning process.

To explore the relational inductive biases expressed within various deep learning methods, we must identify several key ingredients: what are the *entities*, what are the *relations*, and what are the *rules* for composing entities and relations, and computing their implications? In deep learning, the entities and relations are typically expressed as distributed representations, and the rules as neural network function approximators; however, the precise forms of the entities, relations, and rules vary between architectures. To understand these differences between architectures, we can further ask how each supports relational reasoning by probing:

- The arguments to the rule functions (e.g., which entities and relations are provided as input).
- How the rule function is reused, or shared, across the computational graph (e.g., across different entities and relations, across different time or processing steps, etc.).
- How the architecture defines interactions versus isolation among representations (e.g., by applying rules to draw conclusions about related entities, versus processing them separately).

3.2.1 Relational inductive biases in standard deep learning building blocks

Fully connected layers

Perhaps the most common building block is a fully connected layer. Typically implemented as a non-linear vector-valued function of vector inputs, each element, or "unit", of the output vector is the dot product between a weight vector, followed by an added bias term, and finally a non-linearity such as a rectified linear unit (ReLU). As such, the entities are the units in the network, the relations are all-to-all (all units in layer i are connected to all units in layer j), and the rules are specified by the weights and biases. The argument to the rule is the full input signal, there is no reuse, and there is no isolation of information (Figure **??**a). The implicit relational inductive bias in a fully connected layer is thus very weak: all input units can interact to determine any output unit's value, independently across outputs (Table **??**).

Convolutional layers

Another common building block is a convolutional layer. It is implemented by convolving an input vector or tensor with a kernel of the same rank, adding a

bias term, and applying a point-wise non-linearity. The entities here are still individual units (or grid elements, e.g. pixels), but the relations are sparser. The differences between a fully connected layer and a convolutional layer impose some important relational inductive biases: locality and translation invariance (Figure **??**b). Locality reflects that the arguments to the relational rule are those entities in close proximity with one another in the input signal's coordinate space, isolated from distal entities. Translation invariance reflects reuse of the same rule across localities in the input. These biases are very effective for processing natural image data because there is high covariance within local neighborhoods, which diminishes with distance, and because the statistics are mostly stationary across an image (Table **??**).

Recurrent layers

A third common building block is a recurrent layer, which is implemented over a sequence of steps. Here, we can view the inputs and hidden states at each processing step as the entities, and the Markov dependence of one step's hidden state on the previous hidden state and the current input, as the relations. The rule for combining the entities takes a step's inputs and hidden state as arguments to update the hidden state. The rule is reused over each step (Figure **??**c), which reflects the relational inductive bias of temporal invariance (similar to a CNN's translational invariance in space). For example, the outcome of some physical sequence of events should not depend on the time of day. RNNs also carry a bias for locality in the sequence via their Markovian structure (Table **??**).

3.3 Graph networks

3.3.1 Graph network (GN) block

graph neural networks

Models in the *graph neural network* family have been explored in a diverse range of problem domains, across supervised, semi-supervised, unsupervised, and reinforcement learning settings. They have been effective at tasks thought to have rich relational structure, such as

- visual scene understanding tasks and few-shot learning
- learn the dynamics of physical systems and multi-agent systems
- reason about knowledge graphs
- predict the chemical properties of molecules
- predict traffic on roads
- classify and segment images and videos and 3D meshes and point clouds

- classify regions in images
- perform semi-supervised text classification
- machine translation
- combinatorial optimization
- boolean satisfiability

The works cited above are by no means an exhaustive list, but provide a representative cross-section of the breadth of domains for which graph neural networks have proven useful.

graph networks (GN)

We now present our *graph networks (GN)* framework, which defines a class of functions for relational reasoning over graph-structured representations. Note, we avoided using the term "neural" in the "graph network" label to reflect that they can be implemented with functions other than neural networks, though here our focus is on neural network implementations.

The main unit of computation in the GN framework is the GN block, a "graph-to-graph" module which takes a graph as input, performs computations over the structure, and returns a graph as output. As described in Box 3, entities are represented by the graph's nodes, relations by the edges, and system-level properties by global attributes.

Definition of "graph"

node, edge, global attribute

Here we use "graph" to mean a directed, attributed multi-graph with a global attribute. In our terminology, a *node* is denoted as $\mathbf{v}_i$, an *edge* as $\mathbf{e}_k$, and the *global attributes* as $\mathbf{u}$. We also use s_k and r_k to indicate the indices of the sender and receiver nodes (see below), respectively, for edge k. To be more precise, we define these terms as:

Directed:	one-way edges, from a "sender" node to a "receiver" node.
Attribute:	properties that can be encoded as a vector, set, or even another graph.
Attributed:	edges and vertices have attributes associated with them.
Global attribute:	a graph-level attribute.
Multi-graph:	there can be more than one edge between vertices, including self-edges.

graph

Within our GN framework, a *graph* is defined as a 3-tuple $G = (\mathbf{u}; V; E)$. The $\mathbf{u}$ is a *global attribute*; the $V = \{\mathbf{v}_i\}_{i=1:N^v}$ is the *set of nodes* (of cardinality N^v), where each $\mathbf{v}_i$ is a node's attribute. The $E = \{(\mathbf{e}_k; r_k; s_k)\}_{k=1:N^e}$ is the *set of edges* (of cardinality N^e), where each $\mathbf{e}_k$ is the edge's attribute, r_k is the index of the receiver node, and s_k is the index of the sender node.

Algorithm 1 Steps of computation in a full GN block.

function GRAPHNETWORK(E, V, $\mathbf{u}$)
  for $k \in \{1 \dots N^e\}$ **do**
    $\mathbf{e}'_k \leftarrow \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right)$ ▷ 1. Compute updated edge attributes
  end for
  for $i \in \{1 \dots N^n\}$ **do**
    let $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\ k=1:N^e}$
    $\bar{\mathbf{e}}'_i \leftarrow \rho^{e\to v}\left(E'_i\right)$ ▷ 2. Aggregate edge attributes per node
    $\mathbf{v}'_i \leftarrow \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right)$ ▷ 3. Compute updated node attributes
  end for
  let $V' = \{\mathbf{v}'\}_{i=1:N^v}$
  let $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$
  $\bar{\mathbf{e}}' \leftarrow \rho^{e\to u}\left(E'\right)$ ▷ 4. Aggregate edge attributes globally
  $\bar{\mathbf{v}}' \leftarrow \rho^{v\to u}\left(V'\right)$ ▷ 5. Aggregate node attributes globally
  $\mathbf{u}' \leftarrow \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right)$ ▷ 6. Compute updated global attribute
  return $(E', V', \mathbf{u}')$
end function

Internal structure of a GN block

A GN block contains three "update" functions, ϕ, and three "aggregation" functions, ρ,

$$\begin{aligned} \mathbf{e}'_k &= \phi^e\left(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}\right) & \bar{\mathbf{e}}'_i &= \rho^{e\to v}\left(E'_i\right) \\ \mathbf{v}'_i &= \phi^v\left(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}\right) & \bar{\mathbf{e}}' &= \rho^{e\to u}\left(E'\right) \\ \mathbf{u}' &= \phi^u\left(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u}\right) & \bar{\mathbf{v}}' &= \rho^{v\to u}\left(V'\right) \end{aligned} \tag{3.1}$$

where $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\ k=1:N^e}$, $V' = \{\mathbf{v}'_i\}_{i=1:N^v}$, and $E' = \bigcup_i E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$.

The ϕ^e is mapped across all edges to compute per-edge updates, the ϕ^v is mapped across all nodes to compute per-node updates, and the ϕ^u is applied once as the global update. The ρ functions each take a set as input, and reduce it to a single element which represents the aggregated information. Crucially, the ρ functions must be invariant to permutations of their inputs, and should take variable numbers of arguments (e.g., elementwise summation, mean, maximum, etc.).

Computational steps within a GN block

When a graph, G, is provided as input to a GN block, the computations proceed from the edge, to the node, to the global level. Algorithm 1 shows the following steps of computation:

1. ϕ^e is applied per edge, with arguments $(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$, and returns $\mathbf{e}'_k$.

The set of resulting per-edge outputs for each node, i, is, $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i,\ k=1:N^e}$. And $E' = \bigcup_i E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$ is the set of all per-edge outputs.

2. $\rho^{e\to v}$ is applied to E'_i, and aggregates the edge updates for edges that project to vertex i, into $\bar{\mathbf{e}}'_i$, which will be used in the next step's node update.
3. ϕ^v is applied to each node i, to compute an updated node attribute, $\mathbf{v}'_i$. The set of resulting per-node outputs is, $V' = \{\mathbf{v}'_i\}_{i=1:N^v}$.
4. $\rho^{e\to u}$ is applied to E', and aggregates all edge updates, into $\bar{\mathbf{e}}'$, which will then be used in the next step's global update.
5. $\rho^{v\to u}$ is applied to V', and aggregates all node updates, into $\bar{\mathbf{v}}'$, which will then be used in the next step's global update.
6. ϕ^u is applied once per graph, and computes an update for the global attribute, $\mathbf{u}'$.

Relational inductive biases in graph networks

Our GN framework imposes several strong relational inductive biases when used as components in a learning process. First, graphs can express arbitrary relationships among entities, which means the GN's input determines how representations interact and are isolated, rather than those choices being determined by the fixed architecture. For example, the assumption that two entities have a relationship—and thus should interact—is expressed by an edge between the entities' corresponding nodes. Similarly, the absence of an edge expresses the assumption that the nodes have no relationship and should not influence each other directly.

Second, graphs represent entities and their relations as sets, which are invariant to permutations. This means GNs are invariant to the order of these elements[1], which is often desirable. For example, the objects in a scene do not have a natural ordering (see Sec. **??**).

Third, a GN's per-edge and per-node functions are reused across all edges and nodes, respectively. This means GNs automatically support a form of combinatorial generalization (see Section **??**): because graphs are composed of edges, nodes, and global features, a single GN can operate on graphs of different sizes (numbers of edges and nodes) and shapes (edge connectivity).

[1] Note, an ordering can be imposed by encoding the indices in the node or edge attributes, or via the edges themselves (e.g. by encoding a chain or partial ordering).

Chapter 4

Combinatorial Optimization and Reasoning with Graph Neural Networks

Summary of [4].

The inductive bias of GNNs effectively encodes combinatorial and relational input due to their invariance to permutations and awareness of input sparsity.

4.1 Introduction

Intuitively, CO deals with problems that involve optimizing a cost (or objective) function by selecting a subset from a finite set, with the latter encoding constraints on the solution space.

4.1.1 What are the Challenges for Machine Learning?

There are several critical challenges in successfully applying machine learning methods within CO, especially for problems involving graphs. Graphs have no unique representation, i.e., renaming or reordering the nodes does not result in different graphs. Hence, for any machine learning method dealing with graphs, taking into account invariance to permutation is crucial. Combinatorial optimization problem instances, especially those arising from the real world, are large and usually sparse. Hence, the employed machine learning method must be scalable and sparsity-aware. Simultaneously, the employed method has to be expressive enough to detect and exploit the relevant patterns in the given instance or data distribution. The machine learning method should be capable of handling auxiliary information, such as objective and user-defined

constraints. Most of the current machine learning approaches are within the supervised regime. That is, they require a large amount of training data to optimize the parameters of the model. In the context of CO, this means solving many possibly hard problem instances, which might prohibit the application of these approaches in real-world scenarios. Further, the machine learning method has to be able to generalize beyond its training data, e.g., transferring to instances of different sizes.

Overall, there is a trade-off between scalability, expressivity, and generalization, any pair of which might conflict. In summary, the key challenges are:

1. Machine learning methods that operate on graph data have to be *invariant* to node *permutations*. They should also exploit the graph *sparsity*.

2. Models should *distinguish* critical structural *patterns* in the provided data while still *scaling* to large real-world instances.

3. *Side information* in the form of high-dimensional vectors attached to nodes and edges, i.e., modeling objectives and additional information, needs to be considered.

4. Models should be *data-efficient*. That is, they should ideally work without requiring large amounts of labeled data, and they should be transferable to *out-of-sample* instances.

4.1.2 How Do GNNs Address These Challenges?

GNNs have recently emerged as machine learning architectures that (partially) address the challenges above.

The key idea underlying GNNs is to compute a vectorial representation, e.g., a real vector, of each node in the input graph by iteratively aggregating features of neighboring nodes. By parameterizing this aggregation step, the GNN is trained in an end-to-end fashion against a loss function, using (stochastic) first-order optimization techniques to adapt to the given data distribution. The promise here is that the learned vector representation encodes crucial graph structures that help solve a CO problem more efficiently. GNNs are invariant and equivariant by design, i.e., they automatically exploit the invariances or symmetries inherent to the given instance or data distribution. Due to their local nature, by aggregating neighborhood information, GNNs naturally exploit sparsity, leading to more scalable models on sparse inputs. Moreover, although scalability is still an issue, they scale linearly with the number of edges and employed parameters, while taking multi-dimensional node and edge features into account, naturally exploiting cost and objective function information. However, the data-efficiency question is still largely open.

4.2 GNNs for Combinatorial Optimization: The State of the Art

Beyond using standard GNN models for CO, the emerging paradigm of algorithmic reasoning provides new perspectives on designing and training GNNs that satisfy natural invariants and properties, enabling improved generalization and interpretability.

4.2.1 On the Primal Side: Finding Feasible Solutions

We begin by discussing the use of GNNs in improving the solution-finding process in CO. It is natural to wonder whether the primal side of CO is worth exploring when, given sufficient time, exact (or complete) algorithms guarantee finding an optimal solution. The following practical scenarios motivate the need for quickly obtaining high-quality feasible solutions.

a) **Optimality guarantees are often not needed** A practitioner may only be interested in the quality of a feasible solution in absolute terms rather than relative to the (typically unknown) optimal value of a problem instance. To assess a heuristic's suitability in this scenario, one can evaluate it on a set of instances for which the optimal value is known. However, when used on a new problem instance, the heuristic's solution can only be assessed via its (absolute) objective value. This situation arises when the CO problem of interest is practically intractable with an exact solver. For example, many vehicle routing problems admit strong MIP formulations that have an exponential number of variables or constraints, similar to the TSP formulation in Example 2, see (Toth and Vigo, 2014). While such problems may be solved exactly using column or constraint generation (Dror et al., 1994), a heuristic that consistently finds good solutions within a short user-defined time limit may be preferable.

b) **Optimality is desired, but quickly finding a good solution is the priority** Because optimality is still of interest here, one would like to use an exact solver that is focused on the primal side. A common use case is to take a good solution and start analyzing it manually in the current application context while the exact solver keeps running in the background. An early feasible solution allows for fast decision-making, early termination of the solver, or even revisiting the mathematical model with additional constraints that were initially ignored. MIP solvers usually provide a parameter that can be set to emphasize finding solutions quickly; see CPLEX's emphasis switch parameter for an example.[1] Among other measures, these parameters increase the time or iterations allotted

[1] https://www.ibm.com/support/knowledgecenter/SSSA5P_20.1.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/MIPEmphasis.html

to primal heuristics at nodes of the search tree, which improves the odds of finding a good solution early on in the search.

Alternatively, one could also develop a custom, standalone heuristic that is executed first, providing a warm start solution to the exact solver. This simple approach is widely used and addresses both goals a) and b) simultaneously when the heuristic in question is effective for the problem of interest. This can also be done in order to obtain a high-quality first solution for initiating a local search.

4.2.2 Algorithmic Reasoning

Neural networks are traditionally powerful in the interpolation regime, i.e., when we expect the distribution of unseen ("test") inputs to roughly match the distribution of the inputs used to train the network. However, they tend to struggle when extrapolating, i.e., when they are evaluated out-of-distribution. For example, merely increasing the test input size, e.g., the number of nodes in the input graph, is often sufficient to lose most of the training's predictive power.

Extrapolating is a potentially important issue for tackling CO problems with (G)NNs trained end-to-end. As a critical feature of a powerful reasoning system, it should apply to any plausible input, not just ones within the training distribution. Therefore, unless we can accurately foreshadow the kinds of inputs our neural CO approach will be solving, it could be essential to address the issue of out-of-distribution generalization in neural networks meaningfully.

One resurging research direction that holds much promise here is algorithmic reasoning, i.e., directly introducing concepts from classical algorithms (Cormen et al., 2009) into neural network architectures or training regimes, typically by learning how to execute them. Classical algorithms have precisely the kind of favorable properties (strong generalization, compositionality, verifiable correctness) that would be desirable for neural network reasoners. Bringing the two sides closer together can therefore yield the kinds of improvements to performance, generalization, and interpretability that are unlikely to occur through architectural gains alone.

Powered by the rapid development of GNNs, algorithmic reasoning experienced a strong resurgence, tackling combinatorial algorithms of superlinear complexity with graph-structured processing at the core. Initial theoretical analyses (Xu et al., 2019b) demonstrate why this is a good idea, GNNs align with dynamic programming (Bellman, 1966), which is a language in which most algorithms can be expressed. Hence, it is viable that most polynomial-time combinatorial reasoners of interest will be modelable using a GNN. We will now investigate alignment in more detail.

Algorithmic alignment

The concept of algorithmic alignment introduced by Xu et al. (2019b) is central to constructing effective algorithmic reasoners that extrapolate better. Informally, a neural network aligns with an algorithm if that algorithm can be partitioned into several parts, each of which can be "easily" modeled by one of the neural network's modules. Essentially, alignment relies on designing neural networks' components and control flow such that they line up well with the underlying algorithm to be learned from data.

The work of Veličković et al. (2020b) on the neural execution of graph algorithms is among the first to propose algorithmic learning as a first-class citizen and suggests several general-purpose modifications to GNNs to make them stronger combinatorial reasoners.

Perspectives and outlooks

While all of the above dealt with improving the performance of GNNs when reasoning algorithmically, for some combinatorial applications, we require the algorithmic performance to always remain perfect — a trait known as strong generalization (Li et al., 2020). Strong generalization was demonstrated to be possible. That is, neural execution engines (NEEs) (Yan et al., 2020) are capable of maintaining 100% accuracy on various combinatorial tasks by leveraging several low-level constructs, learning individual primitive units of computation, such as addition, multiplication, or argmax, in isolation. Moreover, they employ explicit masking inductive biases and binary representations of inputs. Here, the focus is less on learning the algorithm itself|the dataflow between the computation units is provided in a hard-coded way, allowing for zero-shot transfer of units between related algorithms.

Index

Bibliography

[1] Allamanis, M. An introduction to graph neural networks: Models and applications.

[2] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V. F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gülçehre, Ç., Song, H. F., Ballard, A. J., Gilmer, J., Dahl, G. E., Vaswani, A., Allen, K. R., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y., and Pascanu, R. Relational inductive biases, deep learning, and graph networks. *arXiv preprint* (2018).

[3] Bresson, X. Graph convolutional networks (gcns).

[4] Cappart, Q., Chételat, D., Khalil, E. B., Lodi, A., Morris, C., and Velickovic, P. Combinatorial optimization and reasoning with graph neural networks. *arXiv preprint* (2021).

[5] Hamilton, W. L. Graph representation learning.

[6] Hamilton, W. L. *Graph Representation Learning*, vol. 14. Morgan and Claypool, 2020.

[7] Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications. *arXiv preprint* (2017).

[8] Veličković, P. Graph representation learning for algorithmic reasoning.

[9] Veličković, P. Intro to graph neural networks (ml tech talks).

[10] Veličković, P. Theoretical foundations of graph neural networks.

[11] Velikovi, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. *arXiv preprint* (2020).

[12] Yan, Y., Swersky, K., Koutra, D., Ranganathan, P., and Hashemi, M. Neural execution engines: Learning to execute subroutines. *arXiv preprint* (2020).