Chapter 7

Theoretical Motivations

In this chapter, we will visit some of the theoretical underpinnings of graph neural networks (GNNs). One of the most intriguing aspects of GNNs is that they were independently developed from distinct theoretical motivations. From one perspective, GNNs were developed based on the theory of graph signal processing, as a generalization of Euclidean convolutions to the non-Euclidean graph domain [Bruna et al., 2014]. At the same time, however, neural message passing approaches—which form the basis of most modern GNNs—were proposed by analogy to message passing algorithms for probabilistic inference in graphical models [Dai et al., 2016]. And lastly, GNNs have been motivated in several works based on their connection to the Weisfeiler-Lehman graph isomorphism test [Hamilton et al., 2017b].

This convergence of three disparate areas into a single algorithm framework is remarkable. That said, each of these three theoretical motivations comes with its own intuitions and history, and the perspective one adopts can have a substantial impact on model development. Indeed, it is no accident that we deferred the description of these theoretical motivations until *after* the introduction of the GNN model itself. In this chapter, our goal is to introduce the key ideas underlying these different theoretical motivations, so that an interested reader is free to explore and combine these intuitions and motivations as they see fit.

7.1 GNNs and Graph Convolutions

In terms of research interest and attention, the derivation of GNNs based on connections to graph convolutions is the dominant theoretical paradigm. In this perspective, GNNs arise from the question: How can we generalize the notion of convolutions to general graph-structured data?

7.1.1 Convolutions and the Fourier Transform

In order to generalize the notion of a convolution to graphs, we first must define what we wish to generalize and provide some brief background details. Let f and h be two functions. We can define the general continuous convolution operation $\star$ as

$$(f \star h)(\mathbf{x}) = \int_{\mathbb{R}^d} f(\mathbf{y})h(\mathbf{x}-\mathbf{y})d\mathbf{y}. \tag{7.1}$$

One critical aspect of the convolution operation is that it can be computed by an element-wise product of the *Fourier transforms* of the two functions:

$$(f \star h)(\mathbf{x}) = \mathcal{F}^{-1}\left(\mathcal{F}(f(\mathbf{x})) \circ \mathcal{F}(h(\mathbf{x}))\right), \tag{7.2}$$

where

$$\mathcal{F}(f(\mathbf{x})) = \hat{f}(\mathbf{s}) = \int_{\mathbb{R}^d} f(\mathbf{x})e^{-2\pi\mathbf{x}^\top \mathbf{s}i}d\mathbf{x} \tag{7.3}$$

is the Fourier transform of $f(\mathbf{x})$ and its inverse Fourier transform is defined as

$$\mathcal{F}^{-1}(\hat{f}(\mathbf{s})) = \int_{\mathbb{R}^d} \hat{f}(\mathbf{s})e^{2\pi\mathbf{x}^\top \mathbf{s}i}d\mathbf{s}. \tag{7.4}$$

In the simple case of univariate discrete data over a finite domain $t \in \{0, .., N-1\}$ (i.e., restricting to finite impulse response filters) we can simplify these operations to a discrete circular convolution[1]

$$(f \star_N h)(t) = \sum_{\tau=0}^{N-1} f(\tau)h((t-\tau)_{\text{mod } N}) \tag{7.5}$$

and a discrete Fourier transform (DFT)

$$s_k = \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} f(x_t)e^{-\frac{i2\pi}{N}kt} \tag{7.6}$$

$$= \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} f(x_t)\left(\cos\left(\frac{2\pi}{N}kt\right) - i\sin\left(\frac{2\pi}{N}kt\right)\right) \tag{7.7}$$

where $s_k \in \{s_0..., s_{N-1}\}$ is the Fourier coefficient corresponding to the sequence $(f(x_0), f(x_1), ..., f(x_{N-1}))$. In Equation (7.5) we use the notation $\star_N$ to emphasize that this is a circular convolution defined over the finite domain $\{0, ..., N-1\}$, but we will often omit this subscript for notational simplicity.

Interpreting the (discrete) Fourier transform The Fourier transform essentially tells us how to represent our input signal as a weighted sum of (complex) sinusoidal waves. If we assume that both the input data

[1] For simplicity, we limit ourselves to finite support for both f and h and define the boundary condition using a modulus operator and circular convolution.

and its Fourier transform are real-valued, we can interpret the sequence $[s_0, s_1, ..., s_{N-1}]$ as the coefficients of a Fourier series. In this view, s_k tells us the amplitude of the complex sinusoidal component $e^{-\frac{i2\pi}{N}k}$, which has frequency $\frac{2\pi k}{N}$ (in radians). Often we will discuss *high-frequency components* that have a large k and vary quickly as well as *low-frequency components* that have $k << N$ and vary more slowly. This notion of low and high frequency components will also have an analog in the graph domain, where we will consider signals propagating between nodes in the graph.

In terms of signal processing, we can view the discrete convolution $f \star h$ as a *filtering operation* of the series $(f(x_1), f(x_2), ..., f(x_N))$ by a filter h. Generally, we view the series as corresponding to the values of the signal throughout time, and the convolution operator applies some filter (e.g., a band-pass filter) to modulate this time-varying signal.

One critical property of convolutions, which we will rely on below, is the fact that they are *translation (or shift) equivariant*:

$$f(t+a) \star g(t) = f(t) \star g(t+a) = (f \star g)(t+a). \tag{7.8}$$

This property means that translating a signal and then convolving it by a filter is equivalent to convolving the signal and then translating the result. Note that as a corollary convolutions are also equivariant to the difference operation:

$$\Delta f(t) \star g(t) = f(t) \star \Delta g(t) = \Delta(f \star g)(t), \tag{7.9}$$

where

$$\Delta f(t) = f(t+1) - f(t) \tag{7.10}$$

is the Laplace (i.e., difference) operator on discrete univariate signals.

These notions of filtering and translation equivariance are central to digital signal processing (DSP) and also underlie the intuition of convolutional neural networks (CNNs), which utilize a discrete convolution on two-dimensional data. We will not attempt to cover even a small fraction of the fields of digital signal processing, Fourier analysis, and harmonic analysis here, and we point the reader to various textbooks on these subjects [Grafakos, 2004, Katznelson, 2004, Oppenheim et al., 1999, Rabiner and Gold, 1975].

7.1.2 From Time Signals to Graph Signals

In the previous section, we (briefly) introduced the notions of filtering and convolutions with respect to discrete time-varying signals. We now discuss how we can connect discrete time-varying signals with signals on a graph. Suppose we have a discrete time-varying signal $f(t_0), f(t_2), ..., f(t_{N-1})$. One way of viewing this signal is as corresponding to a chain (or cycle) graph (Figure 7.1), where each point in time t is represented as a node and each function value $f(t)$ represents the signal value at that time/node. Taking this view, it is convenient to represent the signal as a vector $\mathbf{f} \in \mathbb{R}^N$, with each dimension corresponding to

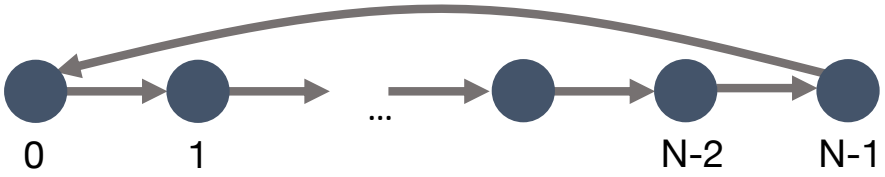

Figure 7.1: Representation of a (cyclic) time-series as a chain graph.

a different node in the chain graph. In other words, we have that $\mathbf{f}[t] = f(t)$ (as a slight abuse of notation). The edges in the graph thus represent how the signal propagates; i.e., the signal propagates forward in time.[2]

One interesting aspect of viewing a time-varying signal as a chain graph is that we can represent operations, such as time-shifts, using the adjacency and Laplacian matrices of the graph. In particular, the adjacency matrix for this chain graph corresponds to the circulant matrix $\mathbf{A}_c$ with

$$\mathbf{A}_c[i,j] = \begin{cases} 1 & \text{if } j = (i+1)_{\text{mod } N} \\ 0 & \text{otherwise,} \end{cases} \tag{7.11}$$

and the (unnormalized) Laplacian $\mathbf{L}_c$ for this graph can be defined as

$$\mathbf{L}_c = \mathbf{I} - \mathbf{A}_c. \tag{7.12}$$

We can then represent time shifts as multiplications by the adjacency matrix,

$$(\mathbf{A}_c\mathbf{f})[t] = \mathbf{f}[(t+1)_{\text{mod } N}], \tag{7.13}$$

and the difference operation by multiplication by the Laplacian,

$$(\mathbf{L}_c\mathbf{f})[t] = \mathbf{f}[t] - \mathbf{f}[(t+1)_{\text{mod } N}]. \tag{7.14}$$

In this way, we can see that there is a close connection between the adjacency and Laplacian matrices of a graph, and the notions of shifts and differences for a signal. Multiplying a signal by the adjacency matrix propagates signals from node to node, and multiplication by the Laplacian computes the difference between a signal at each node and its immediate neighbors.

Given this graph-based view of transforming signals through matrix multiplication, we can similarly represent convolution by a filter h as matrix multiplication on the vector $\mathbf{f}$:

$$(f \star h)(\mathbf{t}) = \sum_{\tau=0}^{N-1} f(\tau)h(\tau - t) \tag{7.15}$$

$$= \mathbf{Q}_h\mathbf{f}, \tag{7.16}$$

[2]Note that we add a connection between the last and first nodes in the chain as a boundary condition to keep the domain finite.

where $\mathbf{Q}_h \in \mathbb{R}^{N \times N}$ is a matrix representation of the convolution operation by filter function h and $\mathbf{f} = [f(t_0), f(t_2), ..., f(t_{N-1})]^\top$ is a vector representation of the function f. Thus, in this view, we consider convolutions that can be represented as a matrix transformation of the signal at each node in the graph.[3] Of course, to have the equality between Equation (7.15) and Equation (7.16), the matrix $\mathbf{Q}_h$ must have some specific properties. In particular, we require that multiplication by this matrix satisfies translation equivariance, which corresponds to commutativity with the circulant adjacency matrix $\mathbf{A}_c$, i.e., we require that

$$\mathbf{A}_c\mathbf{Q}_h = \mathbf{Q}_h\mathbf{A}_c. \tag{7.17}$$

The equivariance to the difference operator is similarly defined as

$$\mathbf{L}_c\mathbf{Q}_h = \mathbf{Q}_h\mathbf{L}_c. \tag{7.18}$$

It can be shown that these requirements are satisfied for a real matrix $\mathbf{Q}_h$ if

$$\mathbf{Q}_h = p_N(\mathbf{A}_c) = \sum_{i=0}^{N-1} \alpha_i \mathbf{A}_c^i, \tag{7.19}$$

i.e., if $\mathbf{Q}_h$ is a polynomial function of the adjacency matrix $\mathbf{A}_c$. In digital signal processing terms, this is equivalent to the idea of representing general filters as polynomial functions of the shift operator [Ortega et al., 2018].[4]

Generalizing to general graphs

We have now seen how shifts and convolutions on time-varying discrete signals can be represented based on the adjacency matrix and Laplacian matrix of a chain graph. Given this view, we can easily generalize these notions to more general graphs.

In particular, we saw that a time-varying discrete signal corresponds to a chain graph and that the notion of translation/difference equivariance corresponds to a commutativity property with adjacency/Laplacian of this chain graph. Thus, we can generalize these notions beyond the chain graph by considering arbitrary adjacency matrices and Laplacians. While the signal simply propagates forward in time in a chain graph, in an arbitrary graph we might have multiple nodes propagating signals to each other, depending on the structure of the adjacency matrix. Based on this idea, we can define convolutional filters on general graphs as matrices $\mathbf{Q}_h$ that commute with the adjacency matrix or the Laplacian.

More precisely, for an arbitrary graph with adjacency matrix $\mathbf{A}$, we can represent convolutional filters as matrices of the following form:

$$\mathbf{Q}_h = \alpha_0\mathbf{I} + \alpha_1\mathbf{A} + \alpha_2\mathbf{A}^2 + ... + \alpha_N\mathbf{A}^N. \tag{7.20}$$

[3] This assumes a real-valued filter h.

[4] Note, however, that there are certain convolutional filters (e.g., complex-valued filters) that cannot be represented in this way.

Intuitively, this gives us a *spatial* construction of a convolutional filter on graphs. In particular, if we multiply a node feature vector $\mathbf{x} \in \mathbb{R}^{|V|}$ by such a convolution matrix $\mathbf{Q_h}$, then we get

$$\mathbf{Q}_h\mathbf{x} = \alpha_0\mathbf{I}\mathbf{x} + \alpha_1\mathbf{A}\mathbf{x} + \alpha_2\mathbf{A}^2\mathbf{x} + ... + \alpha_N\mathbf{A}^N\mathbf{x}, \tag{7.21}$$

which means that the convolved signal $\mathbf{Q}_h\mathbf{x}[u]$ at each node $u \in \mathcal{V}$ will correspond to some mixture of the information in the node's N-hop neighborhood, with the $\alpha_0, ..., \alpha_N$ terms controlling the strength of the information coming from different hops.

We can easily generalize this notion of a graph convolution to higher dimensional node features. If we have a matrix of node features $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}|\times m}$ then we can similarly apply the convolutional filter as

$$\mathbf{Q}_h\mathbf{X} = \alpha_0\mathbf{I}\mathbf{X} + \alpha_1\mathbf{A}\mathbf{X} + \alpha_2\mathbf{A}^2\mathbf{X} + ... + \alpha_N\mathbf{A}^N\mathbf{X}. \tag{7.22}$$

From a signal processing perspective, we can view the different dimensions of the node features as different "channels".

Graph convolutions and message passing GNNs

Equation (7.22) also reveals the connection between the message passing GNN model we introduced in Chapter 5 and graph convolutions. For example, in the basic GNN approach (see Equation 6.5) each layer of message passing essentially corresponds to an application of the simple convolutional filter

$$\mathbf{Q}_h = \mathbf{I} + \mathbf{A} \tag{7.23}$$

combined with some learnable weight matrices and a non-linearity. In general, each layer of message passing GNN architecture aggregates information from a node's local neighborhood and combines this information with the node's current representation (see Equation 5.4). We can view these message passing layers as a generalization of the simple linear filter in Equation (7.23), where we use more complex non-linear functions. Moreover, by stacking multiple message passing layers, GNNs are able to implicitly operate on higher order polynomials of the adjacency matrix.

The adjacency matrix, Laplacian, or a normalized variant? In Equation (7.22) we defined a convolution matrix $\mathbf{Q}_h$ for arbitrary graphs as a polynomial of the adjacency matrix. Defining $\mathbf{Q}_h$ in this way guarantees that our filter commutes with the adjacency matrix, satisfying a generalized notion of *translation equivariance*. However, in general commutativity with the adjacency matrix (i.e., *translation equivariance*) does not necessarily imply commutativity with the Laplacian $\mathbf{L} = \mathbf{D} - \mathbf{A}$ (or any of its normalized variants). In this special case of the chain graph, we were able to define filter matrices $\mathbf{Q}_h$ that simultaneously commute with both $\mathbf{A}$ and $\mathbf{L}$, but for more general graphs we have a choice to make in terms

of whether we define convolutions based on the adjacency matrix or some version of the Laplacian. Generally, there is no "right" decision in this case, and there can be empirical trade-offs depending on the choice that is made. Understanding the theoretical underpinnings of these trade-offs is an open area of research [Ortega et al., 2018].

In practice researchers often use the symmetric normalized Laplacian $\mathbf{L}_{\text{sym}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{L}\mathbf{D}^{-\frac{1}{2}}$ or the symmetric normalized adjacency matrix $\mathbf{A}_{\text{sym}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ to define convolutional filters. There are two reasons why these symmetric normalized matrices are desirable. First, both these matrices have bounded spectrums, which gives them desirable numerical stability properties. In addition—and perhaps more importantly—these two matrices are *simultaneously diagonalizable*, which means that they share the same eigenvectors. In fact, one can easily verify that there is a simple relationship between their eigendecompositions, since

$$\begin{aligned} &\mathbf{L}_{\text{sym}} = \mathbf{I} - \mathbf{A}_{\text{sym}} \\ &\Rightarrow \\ &\mathbf{L}_{\text{sym}} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top \qquad \mathbf{A}_{\text{sym}} = \mathbf{U}(\mathbf{I} - \mathbf{\Lambda})\mathbf{U}^\top, \end{aligned} \tag{7.24}$$

where $\mathbf{U}$ is the shared set of eigenvectors and $\mathbf{\Lambda}$ is the diagonal matrix containing the Laplacian eigenvalues. This means that defining filters based on one of these matrices implies commutativity with the other, which is a very convenient and desirable property.

7.1.3 Spectral Graph Convolutions

We have seen how to generalize the notion of a signal and a convolution to the graph domain. We did so by analogy to some important properties of discrete convolutions (e.g., translation equivariance), and this discussion led us to the idea of representing graph convolutions as polynomials of the adjacency matrix (or the Laplacian). However, one key property of convolutions that we ignored in the previous subsection is the relationship between convolutions and the Fourier transform. In this section, we will thus consider the notion of a *spectral convolution* on graphs, where we construct graph convolutions via an extension of the Fourier transform to graphs. We will see that this spectral perspective recovers many of the same results we previously discussed, while also revealing some more general notions of a graph convolution.

The Fourier transform and the Laplace operator

To motivate the generalization of the Fourier transform to graphs, we rely on the connection between the Fourier transform and the Laplace (i.e., difference) operator. We previously saw a definition of the Laplace operator Δ in the case of a simple discrete time-varying signal (Equation 7.10) but this operator can

be generalized to apply to arbitrary smooth functions $f : \mathbb{R}^d \to \mathbb{R}$ as

$$\Delta f(\mathbf{x}) = \nabla^2 f(\mathbf{x}) \tag{7.25}$$

$$= \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x^2}. \tag{7.26}$$

This operator computes the *divergence* ∇ of the *gradient* $\nabla f(\mathbf{x})$. Intuitively, the Laplace operator tells us the average difference between the function value at a point and function values in the neighboring regions surrounding this point.

In the discrete time setting, the Laplace operator simply corresponds to the difference operator (i.e., the difference between consecutive time points). In the setting of general discrete graphs, this notion corresponds to the Laplacian, since by definition

$$(\mathbf{L}\mathbf{x})[i] = \sum_{j \in \mathcal{V}} \mathbf{A}[i, j](\mathbf{x}[i] - \mathbf{x}[j]), \tag{7.27}$$

which measures the difference between the value of some signal $\mathbf{x}[i]$ at a node i and the signal values of all of its neighbors. In this way, we can view the Laplacian matrix as a discrete analog of the Laplace operator, since it allows us to quantify the difference between the value at a node and the values at that node's neighbors.

Now, an extremely important property of the Laplace operator is that its eigenfunctions correspond to the complex exponentials. That is,

$$-\Delta(e^{2\pi i s t}) = -\frac{\partial^2 (e^{2\pi i s t})}{\partial t^2} = (2\pi s)^2 e^{2\pi i s t}, \tag{7.28}$$

so the eigenfunctions of Δ are the same complex exponentials that make up the modes of the frequency domain in the Fourier transform (i.e., the sinusoidal plane waves), with the corresponding eigenvalue indicating the frequency. In fact, one can even verify that the eigenvectors $\mathbf{u}_1, ..., \mathbf{u}_n$ of the circulant Laplacian $\mathbf{L}_c \in \mathbb{R}^{n \times n}$ for the chain graph are $\mathbf{u}_j = \frac{1}{\sqrt{n}}[1, \omega_j, \omega_j^2, ..., \omega_j^n]$ where $\omega_j = e^{\frac{2\pi j}{n}}$.

The graph Fourier transform

The connection between the eigenfunctions of the Laplace operator and the Fourier transform allows us to generalize the Fourier transform to arbitrary graphs. In particular, we can generalize the notion of a Fourier transform by considering the eigendecomposition of the general graph Laplacian:

$$\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^\top, \tag{7.29}$$

where we define the eigenvectors $\mathbf{U}$ to be the *graph Fourier modes*, as a graph-based notion of Fourier modes. The matrix $\mathbf{\Lambda}$ is assumed to have the corresponding eigenvalues along the diagonal, and these eigenvalues provide a graph-based notion of different frequency values. In other words, since the eigenfunctions of

the general Laplace operator correspond to the Fourier modes—i.e., the complex exponentials in the Fourier series—we define the Fourier modes for a general graph based on the eigenvectors of the graph Laplacian.

Thus, the Fourier transform of signal (or function) $\mathbf{f} \in \mathbb{R}^{|\mathcal{V}|}$ on a graph can be computed as

$$\mathbf{s} = \mathbf{U}^\top \mathbf{f} \tag{7.30}$$

and its inverse Fourier transform computed as

$$\mathbf{f} = \mathbf{U}\mathbf{s}. \tag{7.31}$$

Graph convolutions in the spectral domain are defined via point-wise products in the transformed Fourier space. In other words, given the graph Fourier coefficients $\mathbf{U}^\top \mathbf{f}$ of a signal $\mathbf{f}$ as well as the graph Fourier coefficients $\mathbf{U}^\top \mathbf{h}$ of some filter $\mathbf{h}$, we can compute a graph convolution via element-wise products as

$$\mathbf{f} \star_{\mathcal{G}} \mathbf{h} = \mathbf{U}\left(\mathbf{U}^\top \mathbf{f} \circ \mathbf{U}^\top \mathbf{h}\right), \tag{7.32}$$

where $\mathbf{U}$ is the matrix of eigenvectors of the Laplacian $\mathbf{L}$ and where we have used $\star_{\mathcal{G}}$ to denote that this convolution is specific to a graph $\mathcal{G}$.

Based on Equation (7.32), we can represent convolutions in the spectral domain based on the graph Fourier coefficients $\theta_h = \mathbf{U}^\top \mathbf{h} \in \mathbb{R}^{|\mathcal{V}|}$ of the function h. For example, we could learn a *non-parametric* filter by directly optimizing θ_h and defining the convolution as

$$\mathbf{f} \star_{\mathcal{G}} \mathbf{h} = \mathbf{U}\left(\mathbf{U}^\top \mathbf{f} \circ \theta_h\right) \tag{7.33}$$

$$= (\mathbf{U}\text{diag}(\theta_h)\mathbf{U}^\top)\mathbf{f} \tag{7.34}$$

where $\text{diag}(\theta_h)$ is matrix with the values of θ_h on the diagonal. However, a filter defined in this non-parametric way has no real dependency on the structure of the graph and may not satisfy many of the properties that we want from a convolution. For example, such filters can be arbitrarily *non-local*.

To ensure that the spectral filter θ_h corresponds to a meaningful convolution on the graph, a natural solution is to parameterize θ_h based on the eigenvalues of the Laplacian. In particular, we can define the spectral filter as $p_N(\mathbf{\Lambda})$, so that it is a degree N polynomial of the eigenvalues of the Laplacian. Defining the spectral convolution in this way ensures our convolution commutes with the Laplacian, since

$$\mathbf{f} \star_{\mathcal{G}} \mathbf{h} = (\mathbf{U}p_N(\mathbf{\Lambda})\mathbf{U}^\top)\mathbf{f} \tag{7.35}$$

$$= p_N(\mathbf{L})\mathbf{f}. \tag{7.36}$$

Moreover, this definition ensures a notion of locality. If we use a degree k polynomial, then we ensure that the filtered signal at each node depends on information in its k-hop neighborhood.

Thus, in the end, deriving graph convolutions from the spectral perspective, we can recover the key idea that graph convolutions can be represented as

polynomials of the Laplacian (or one of its normalized variants). However, the spectral perspective also reveals more general strategies for defining convolutions on graphs.

Interpreting the Laplacian eigenvectors as frequencies In the standard Fourier transform we can interpret the Fourier coefficients as corresponding to different frequencies. In the general graph case, we can no longer interpret the graph Fourier transform in this way. However, we can still make analogies to high frequency and low frequency components. In particular, we can recall that the eigenvectors $\mathbf{u}_i, i = 1, ..., |\mathcal{V}|$ of the Laplacian solve the minimization problem:

$$\min_{\mathbf{u}_i \in \mathbb{R}^{|\mathcal{V}|}: \mathbf{u}_i \perp \mathbf{u}_j \forall j < i} \frac{\mathbf{u}_i^\top \mathbf{L} \mathbf{u}_i}{\mathbf{u}_i^\top \mathbf{u}_j} \tag{7.37}$$

by the Rayleigh-Ritz Theorem. And we have that

$$\mathbf{u}_i^\top \mathbf{L} \mathbf{u}_i = \frac{1}{2} \sum_{u,v \in \mathcal{V}} \mathbf{A}[u, v](\mathbf{u}_i[u] - \mathbf{u}_i[v])^2 \tag{7.38}$$

by the properties of the Laplacian discussed in Chapter 1. Together these facts imply that the smallest eigenvector of the Laplacian corresponds to a signal that varies from node to node by the least amount on the graph, the second smallest eigenvector corresponds to a signal that varies the second smallest amount, and so on. Indeed, we leveraged these properties of the Laplacian eigenvectors in Chapter 1 when we performed spectral clustering. In that case, we showed that the Laplacian eigenvectors can be used to assign nodes to communities so that we minimize the number of edges that go between communities. We can now interpret this result from a signal processing perspective: the Laplacian eigenvectors define signals that vary in a smooth way across the graph, with the smoothest signals indicating the coarse-grained community structure of the graph.

7.1.4 Convolution-Inspired GNNs

The previous subsections generalized the notion of convolutions to graphs. We saw that basic convolutional filters on graphs can be represented as polynomials of the (normalized) adjacency matrix or Laplacian. We saw both spatial and spectral motivations of this fact, and we saw how the spectral perspective can be used to define more general forms of graph convolutions based on the graph Fourier transform. In this section, we will briefly review how different GNN models have been developed and inspired based on these connections.

Purely convolutional approaches

Some of the earliest work on GNNs can be directly mapped to the graph convolution definitions of the previous subsections. The key idea in these approaches is that they use either Equation (7.34) or Equation (7.35) to define a convolutional layer, and a full model is defined by stacking and combining multiple convolutional layers with non-linearities. For example, in early work Bruna et al. [2014] experimented with the non-parametric spectral filter (Equation 7.34) as well as a parametric spectral filter (Equation 7.35), where they defined the polynomial $p_N(\mathbf{\Lambda})$ via a cubic spline approach. Following on this work, Defferrard et al. [2016] defined convolutions based on Equation 7.35 and defined $p_N(\mathbf{L})$ using *Chebyshev polynomials*. This approach benefits from the fact that Chebyshev polynomials have an efficient recursive formulation and have various properties that make them suitable for polynomial approximation [Mason and Handscomb, 2002]. In a related approach, Liao et al. [2019b] learn polynomials of the Laplacian based on the Lanczos algorithm.

There are also approaches that go beyond real-valued polynomials of the Laplacian (or the adjacency matrix). For example, Levie et al. [2018] consider *Cayley polynomials* of the Laplacian and Bianchi et al. [2019] consider *ARMA filters*. Both of these approaches employ more general parametric rational complex functions of the Laplacian (or the adjacency matrix).

Graph convolutional networks and connections to message passing

In their seminal work, Kipf and Welling [2016a] built off the notion of graph convolutions to define one of the most popular GNN architectures, commonly known as the graph convolutional network (GCN). The key insight of the GCN approach is that we can build powerful models by stacking very simple graph convolutional layers. A basic GCN layer is defined in Kipf and Welling [2016a] as

$$\mathbf{H}^{(k)} = \sigma\left(\tilde{\mathbf{A}}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}\right), \tag{7.39}$$

where $\tilde{\mathbf{A}} = (\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}(\mathbf{I}+\mathbf{A})(\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}$ is a normalized variant of the adjacency matrix (with self-loops) and $\mathbf{W}^{(k)}$ is a learnable parameter matrix. This model was initially motivated as a combination of a simple graph convolution (based on the polynomial $\mathbf{I}+\mathbf{A}$), with a learnable weight matrix, and a non-linearity.

As discussed in Chapter 5 we can also interpret the GCN model as a variation of the basic GNN message passing approach. In general, if we consider combining a simple graph convolution defined via the polynomial $\mathbf{I}+\mathbf{A}$ with non-linearities and trainable weight matrices we recover the basic GNN:

$$\mathbf{H}^{(k)} = \sigma\left(\mathbf{A}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}_{\text{neigh}} + \mathbf{H}^{(k-1)}\mathbf{W}^{(k)}_{\text{self}}\right). \tag{7.40}$$

In other words, a simple graph convolution based on $\mathbf{I}+\mathbf{A}$ is equivalent to aggregating information from neighbors and combining that with information

from the node itself. Thus we can view the notion of message passing as corresponding to a simple form of graph convolutions combined with additional trainable weights and non-linearities.

Over-smoothing as a low-pass convolutional filter In Chapter 5 we introduced the problem of *over-smoothing* in GNNs. The intuitive idea in over-smoothing is that after too many rounds of message passing, the embeddings for all nodes begin to look identical and are relatively uninformative. Based on the connection between message-passing GNNs and graph convolutions, we can now understand over-smoothing from the perspective of graph signal processing.

The key intuition is that stacking multiple rounds of message passing in a basic GNN is analogous to applying a low-pass convolutional filter, which produces a smoothed version of the input signal on the graph. In particular, suppose we simplify a basic GNN (Equation 7.40) to the following update equation:

$$\mathbf{H}^{(k)} = \mathbf{A}_{\text{sym}}\mathbf{H}^{(k-1)}\mathbf{W}^{(k)}. \tag{7.41}$$

Compared to the basic GNN in Equation (7.40), we have simplified the model by removing the non-linearity and removing addition of the "self" embeddings at each message-passing step. For mathematical simplicity and numerical stability, we will also assume that we are using the symmetric normalized adjacency matrix $\mathbf{A}_{\text{sym}} = \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ rather than the unnormalized adjacency matrix. This model is similar to the simple GCN approach proposed in Kipf and Welling [2016a] and essentially amounts to taking the average over the neighbor embeddings at each round of message passing.

Now, it is easy to see that after K rounds of message passing based on Equation (7.41), we will end up with a representation that depends on the Kth power of the adjacency matrix:

$$\mathbf{H}^{(K)} = \mathbf{A}^{K}_{\text{sym}}\mathbf{X}\mathbf{W}, \tag{7.42}$$

where $\mathbf{W}$ is some linear operator and $\mathbf{X}$ is the matrix of input node features. To understand the connection between over-smoothing and convolutional filters, we just need to recognize that the multiplication $\mathbf{A}^{K}_{\text{sym}}\mathbf{X}$ of the input node features by a high power of the adjacency matrix can be interpreted as convolutional filter based on the lowest-frequency signals of the graph Laplacian.

For example, suppose we use a large enough value of K such that we have reached the a fixed point of the following recurrence:

$$\mathbf{A}_{\text{sym}}\mathbf{H}^{(K)} = \mathbf{H}^{(K)}. \tag{7.43}$$

One can verify that this fixed point is attainable when using the normalized adjacency matrix, since the dominant eigenvalue of $\mathbf{A}_{\text{sym}}$ is equal to one.

We can see that at this fixed point, all the node features will have converged to be completely defined by the dominant eigenvector of $\mathbf{A}_{\text{sym}}$, and more generally, higher powers of $\mathbf{A}_{\text{sym}}$ will emphasize the largest eigenvalues of this matrix. Moreover, we know that the largest eigenvalues of $\mathbf{A}_{\text{sym}}$ correspond to the smallest eigenvalues of its counterpart, the symmetric normalized Laplacian $\mathbf{L}_{\text{sym}}$ (e.g., see Equation 7.24). Together, these facts imply that multiplying a signal by high powers of $\mathbf{A}_{\text{sym}}$ corresponds to a convolutional filter based on the *lowest* eigenvalues (or frequencies) of $\mathbf{L}_{\text{sym}}$, i.e., it produces a low-pass filter!

Thus, we can see from this simplified model that stacking many rounds of message passing leads to convolutional filters that are low-pass, and—in the worst case—these filters simply converge all the node representations to constant values within connected components on the graph (i.e., the "zero-frequency" of the Laplacian).

Of course, in practice we use more complicated forms of message passing, and this issue is partially alleviated by including each node's previous embedding in the message-passing update step. Nonetheless, it is instructive to understand how stacking "deeper" convolutions on graphs in a naive way can actually lead to simpler, rather than more complex, convolutional filters.

GNNs without message passing

Inspired by connections to graph convolutions, several recent works have also proposed to simplify GNNs by removing the iterative message passing process. In these approaches, the models are generally defined as

$$\mathbf{Z} = \text{MLP}_\theta\left(f(\mathbf{A})\text{MLP}_\phi(\mathbf{X})\right), \tag{7.44}$$

where $f : \mathbb{R}^{\mathcal{V}|\times|\mathcal{V}|} \to \mathbb{R}^{\mathcal{V}|\times|\mathcal{V}|}$ is some deterministic function of the adjacency matrix $\mathbf{A}$, MLP denotes a dense neural network, $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}|\times m}$ is the matrix of input node features, and $\mathbf{Z} \in \mathbb{R}^{|\mathcal{V}|\times d}$ is the matrix of learned node representations. For example, in Wu et al. [2019], they define

$$f(\mathbf{A}) = \tilde{\mathbf{A}}^k, \tag{7.45}$$

where $\tilde{\mathbf{A}} = (\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}(\mathbf{A}+\mathbf{I})(\mathbf{D}+\mathbf{I})^{-\frac{1}{2}}$ is the symmetric normalized adjacency matrix (with self-loops added). In a closely related work Klicpera et al. [2019] defines f by analogy to the personalized PageRank algorithm as[5]

$$f(\mathbf{A}) = \alpha(\mathbf{I} - (1-\alpha)\tilde{\mathbf{A}})^{-1} \tag{7.46}$$

$$= \alpha\sum_{k=0}^{\infty}\left(\mathbf{I} - \alpha\tilde{\mathbf{A}}\right)^k. \tag{7.47}$$

[5]Note that the equality between Equations (7.46) and (7.47) requires that the dominant eigenvalue of $(\mathbf{I} - \alpha\mathbf{A})$ is bounded above by 1. In practice, Klicpera et al. [2019] use power iteration to approximate the inversion in Equation (7.46).

The intuition behind these approaches is that we often do not need to interleave trainable neural networks with graph convolution layers. Instead, we can simply use neural networks to learn feature transformations at the beginning and end of the model and apply a deterministic convolution layer to leverage the graph structure. These simple models are able to outperform more heavily parameterized message passing models (e.g., GATs or `GraphSAGE`) on many classification benchmarks.

There is also increasing evidence that using the symmetric normalized adjacency matrix with self-loops leads to effective graph convolutions, especially in this simplified setting without message passing. Both Wu et al. [2019] and Klicpera et al. [2019] found that convolutions based on $\tilde{\mathbf{A}}$ achieved the best empirical performance. Wu et al. [2019] also provide theoretical support for these results. They prove that adding self-loops shrinks the spectrum of corresponding graph Laplacian by reducing the magnitude of the dominant eigenvalue. Intuitively, adding self-loops decreases the influence of far-away nodes and makes the filtered signal more dependent on local neighborhoods on the graph.

7.2 GNNs and Probabilistic Graphical Models

GNNs are well-understood and well-motivated as extensions of convolutions to graph-structured data. However, there are alternative theoretical motivations for the GNN framework that can provide interesting and novel perspectives. One prominent example is the motivation of GNNs based on connections to variational inference in probabilistic graphical models (PGMs).

In this probabilistic perspective, we view the embeddings $\mathbf{z}_u, \forall u \in \mathcal{V}$ for each node as *latent variables* that we are attempting to infer. We assume that we observe the graph structure (i.e., the adjacency matrix, $\mathbf{A}$) and the input node features, $\mathbf{X}$, and our goal is to infer the underlying latent variables (i.e., the embeddings $\mathbf{z}_v$) that can explain this observed data. The message passing operation that underlies GNNs can then be viewed as a neural network analogue of certain message passing algorithms that are commonly used for variational inference to infer distributions over latent variables. This connection was first noted by Dai et al. [2016], and much of the proceeding discussions is based closely on their work.

Note that the presentation in this section assumes a substantial background in PGMs, and we recommend Wainwright and Jordan [2008] as a good resource for the interested reader. However, we hope and expect that even a reader without any knowledge of PGMS can glean useful insights from the following discussions.

7.2.1 Hilbert Space Embeddings of Distributions

To understand the connection between GNNs and probabilistic inference, we first (briefly) introduce the notion of embedding distributions in Hilbert spaces [Smola et al., 2007]. Let $p(\mathbf{x})$ denote a probability density function defined

over the random variable $\mathbf{x} \in \mathbb{R}^m$. Given an arbitrary (and possibly infinite dimensional) feature map $\phi : \mathbb{R}^m \to \mathcal{R}$, we can represent the density $p(\mathbf{x})$ based on its expected value under this feature map:

$$\mu_{\mathbf{x}} = \int_{\mathbb{R}^m} \phi(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}. \tag{7.48}$$

The key idea with Hilbert space embeddings of distributions is that Equation (7.48) will be *injective*, as long as a suitable feature map ϕ is used. This means that $\mu_{\mathbf{x}}$ can serve as a *sufficient statistic* for $p(\mathbf{x})$, and any computations we want to perform on $p(\mathbf{x})$ can be equivalently represented as functions of the embedding $\mu_{\mathbf{x}}$. A well-known example of a feature map that would guarantee this injective property is the feature map induced by the Gaussian radial basis function (RBF) kernel [Smola et al., 2007].

The study of Hilbert space embeddings of distributions is a rich area of statistics. In the context of the connection to GNNs, however, the key takeaway is simply that we can represent distributions $p(\mathbf{x})$ as embeddings μ_x in some feature space. We will use this notion to motivate the GNN message passing algorithm as a way of learning embeddings that represent the distribution over node latents $p(\mathbf{z}_v)$.

7.2.2 Graphs as Graphical Models

Taking a probabilistic view of graph data, we can assume that the graph structure we are given defines the *dependencies* between the different nodes. Of course, we usually interpret graph data in this way. Nodes that are connected in a graph are generally assumed to be related in some way. However, in the probabilistic setting, we view this notion of dependence between nodes in a formal, probabilistic way.

To be precise, we say that a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ defines a Markov random field:

$$p(\{\mathbf{x}_v\}, \{\mathbf{z}_v\}) \propto \prod_{v \in V} \Phi(\mathbf{x}_v, \mathbf{z}_v) \prod_{(u,v) \in \mathcal{E}} \Psi(\mathbf{z}_u, \mathbf{z}_v), \tag{7.49}$$

where Φ and Ψ are non-negative *potential functions*, and where we use $\{\mathbf{x}_v\}$ as a shorthand for the set $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$. Equation (7.49) says that the distribution $p(\{\mathbf{x}_v\}, \{\mathbf{z}_v\})$ over node features and node embeddings factorizes according to the graph structure. Intuitively, $\Phi(\mathbf{x}_v, \mathbf{z}_v)$ indicates the likelihood of a node feature vector $\mathbf{x}_v$ given its latent node embedding $\mathbf{z}_v$, while Ψ controls the dependency between connected nodes. We thus assume that node features are determined by their latent embeddings, and we assume that the latent embeddings for connected nodes are dependent on each other (e.g., connected nodes might have similar embeddings).

In the standard probabilistic modeling setting, Φ and Ψ are usually defined as parametric functions based on domain knowledge, and, most often, these functions are assumed to come from the exponential family to ensure tractability [Wainwright and Jordan, 2008]. In our presentation, however, we are agnostic to

the exact form of Φ and Ψ, and we will seek to *implicitly* learn these functions by leveraging the Hilbert space embedding idea discussed in the previous section.

7.2.3 Embedding mean-field inference

Given the Markov random field defined by Equation (7.49), our goal is to infer the distribution of latent embeddings $p(\mathbf{z}_v)$ for all the nodes $v \in V$, while also implicitly learning the potential functions Φ and Ψ. In more intuitive terms our goal is to infer latent representations for all the nodes in the graph that can explain the dependencies between the observed node features.

In order to do so, a key step is computing the posterior $p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})$, i.e., computing the likelihood of a particular set of latent embeddings given the observed features. In general, computing this posterior is computationally intractable—even if Φ and Ψ are known and well-defined—so we must resort to approximate methods.

One popular approach—which we will leverage here—is to employ *mean-field variational inference*, where we approximate the posterior using some functions q_v based on the assumption:

$$p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\}) \approx q(\{\mathbf{z}_v\}) = \prod_{v \in \mathcal{V}} q_v(\mathbf{z}_v), \tag{7.50}$$

where each q_v is a valid density. The key intuition in mean-field inference is that we assume that the posterior distribution over the latent variables factorizes into $\mathcal{V}$ independent distributions, one per node.

To obtain approximating q_v functions that are optimal in the mean-field approximation, the standard approach is to minimize the Kullback–Leibler (KL) divergence between the approximate posterior and the true posterior:

$$\mathrm{KL}(q(\{\mathbf{z}_v\})|\{p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\}) = \int_{(\mathbb{R}^d)^{\mathcal{V}}} \prod_{v \in \mathcal{V}} q(\{\mathbf{z}_v\}) \log \left(\frac{\prod_{v \in \mathcal{V}} q(\{\mathbf{z}_v\})}{p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})} \right) \prod_{v \in \mathcal{V}} d\mathbf{z}_v. \tag{7.51}$$

The KL divergence is one canonical way of measuring the distance between probability distributions, so finding q_v functions that minimize Equation (7.51) gives an approximate posterior that is as close as possible to the true posterior under the mean-field assumption. Of course, directly minimizing Equation (7.51) is impossible, since evaluating the KL divergence requires knowledge of the true posterior.

Luckily, however, techniques from variational inference can be used to show that $q_v(\mathbf{z}_v)$ that minimize the KL must satisfy the following fixed point equations:

$$\log(q(\mathbf{z}_v)) = c_v + \log(\Phi(\mathbf{x}_v, \mathbf{z}_v)) + \sum_{u \in \mathcal{N}(v)} \int_{\mathbb{R}^d} q_u(\mathbf{z}_u) \log\left(\Psi(\mathbf{z}_u, \mathbf{z}_v)\right) d\mathbf{z}_u, \tag{7.52}$$

where c_v is a constant that does not depend on $q_v(\mathbf{z}_v)$ or $\mathbf{z}_v$. In practice, we can approximate this fixed point solution by initializing some initial guesses $q_v^{(t)}$ to valid probability distributions and iteratively computing

$$\log\left(q_v^{(t)}(\mathbf{z}_v)\right) = c_v + \log(\Phi(\mathbf{x}_v, \mathbf{z}_v)) + \sum_{u \in \mathcal{N}(v)} \int_{\mathbb{R}^d} q_u^{(t-1)}(\mathbf{z}_u) \log\left(\Psi(\mathbf{z}_u, \mathbf{z}_v)\right) d\mathbf{z}_u. \tag{7.53}$$

The justification behind Equation (7.52) is beyond the scope of this book. For the purposes of this book, however, the essential ideas are the following:

1. We can approximate the true posterior $p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})$ over the latent embeddings using the mean-field assumption, where we assume that the posterior factorizes into $|\mathcal{V}|$ independent distributions $p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\}) \approx \prod_{v \in \mathcal{V}} q_v(\mathbf{z}_v)$.
2. The optimal approximation under the mean-field assumption is given by the fixed point in Equation (7.52), where the approximate posterior $q_v(\mathbf{z}_v)$ for each latent node embedding is a function of (i) the node's feature $\mathbf{z}_x$ and (ii) the marginal distributions $q_u(\mathbf{z})_u, \forall u \in \mathcal{N}(v)$ of the node's neighbors' embeddings.

At this point the connection to GNNs begins to emerge. In particular, if we examine the fixed point iteration in Equation (7.53), we see that the updated marginal distribution $q_v^{(t)}(\mathbf{z}_v)$ is a function of the node features $\mathbf{x}_v$ (through the potential function Φ) as well as function of the set of neighbor marginals $\{q_u^{(t-1)}(\mathbf{z}_u), \forall u \in \mathcal{N}(v)\}$ from the previous iteration (through the potential function Ψ). This form of message passing is highly analogous to the message passing in GNNs! At each step, we are updating the values at each node based on the set of values in the node's neighborhood. The key distinction is that the mean-field message passing equations operate over *distributions* rather than *embeddings*, which are used in the standard GNN message passing.

We can make the connection between GNNs and mean-field inference even tighter by leveraging the Hilbert space embeddings that we introduced in Section 7.2.1. Suppose we have some injective feature map ϕ and can represent all the marginals $q_v(\mathbf{z}_v)$ as embeddings

$$\mu_v = \int_{\mathbb{R}^d} q_v(\mathbf{z}_v)\phi(\mathbf{z}_v)d\mathbf{z}_v \in \mathbb{R}^d. \tag{7.54}$$

With these representations, we can re-write the fixed point iteration in Equation (7.52) as

$$\mu_v^{(t)} = \mathbf{c} + f(\mu_v^{(t-1)}, \mathbf{x}_v, \{\mu_u, \forall u \in \mathcal{N}(v)\} \tag{7.55}$$

where f is a vector-valued function. Notice that f *aggregates* information from the set of neighbor embeddings (i.e., $\{\mu_u, \forall u \in \mathcal{N}(v)\}$ and *updates* the node's current representation (i.e., $\mu_v^{(t-1)}$) using this aggregated data. In this way, we can see that embedded mean-field inference exactly corresponds to a form of neural message passing over a graph!

Now, in the usual probabilistic modeling scenario, we would define the potential functions Φ and Ψ, as well as the feature map ϕ, using some domain knowledge. And given some Φ, Ψ, and ψ we could then try to analytically derive the f function in Equation (7.55) that would allow us to work with an embedded version of mean field inference. However, as an alternative, we can simply try to learn embeddings μ_v in and end-to-end fashion using some supervised signals, and we can define f to be an arbitrary neural network. In other words, rather than specifying a concrete probabilistic model, we can simply learn embeddings μ_v that *could* correspond to *some* probabilistic model. Based on this idea, Dai et al. [2016] define f in an analogous manner to a basic GNN as

$$\mu_v^{(t)} = \sigma\left(\mathbf{W}_{\text{self}}^{(t)}\mathbf{x}_v + \mathbf{W}_{\text{neigh}}^{(t)} \sum_{u \in \mathcal{N}(v)} \mu_u^{(t-1)}\right). \tag{7.56}$$

Thus, at each iteration, the updated Hilbert space embedding for node v is a function of its neighbors' embeddings as well as its feature inputs. And, as with a basic GNN, the parameters $\mathbf{W}_{\text{self}}^{(t)}$ and $\mathbf{W}_{\text{neigh}}^{(t)}$ of the update process can be trained via gradient descent on any arbitrary task loss.

7.2.4 GNNs and PGMs More Generally

In the previous subsection, we gave a brief introduction to how a basic GNN model can be derived as an embedded form of mean field inference—a connection first outlined by Dai et al. [2016]. There are, however, further ways to connect PGMs and GNNs. For example, different variants of message passing can be derived based on different approximate inference algorithms (e.g., Bethe approximations as discussed in Dai et al. [2016]), and there are also several works which explore how GNNs can be integrated more generally into PGM models [Qu et al., 2019, Zhang et al., 2020]. In general, the connections between GNNs and more traditional statistical relational learning is a rich area with vast potential for new developments.

7.3 GNNs and Graph Isomorphism

We have now seen how GNNs can be motivated based on connections to graph signal processing and probabilistic graphical models. In this section, we will turn our attention to our third and final theoretical perspective on GNNs: the motivation of GNNs based on connections to graph isomorphism testing.

As with the previous sections, here we will again see how the basic GNN can be derived as a neural network variation of an existing algorithm—in this case the Weisfieler-Lehman (WL) isomorphism algorithm. However, in addition to motivating the GNN approach, connections to isomorphism testing will also provide us with tools to analyze the power of GNNs in a formal way.

7.3.1 Graph Isomorphism

Testing for graph isomorphism is one of the most fundamental and well-studied tasks in graph theory. Given a pair of graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, the goal of graph isomorphism testing is to declare whether or not these two graphs are *isomorphic*. In an intuitive sense, two graphs being isomorphic means that they are essentially identical. Isomorphic graphs represent the exact same graph structure, but they might differ only in the ordering of the nodes in their corresponding adjacency matrices. Formally, if we have two graphs with adjacency matrices $\mathbf{A}_1$ and $\mathbf{A}_2$, as well as node features $\mathbf{X}_1$ and $\mathbf{X}_2$, we say that two graphs are isomorphic if and only if there exists a permutation matrix $\mathbf{P}$ such that

$$\mathbf{P}\mathbf{A}_1\mathbf{P}^\top = \mathbf{A}_2 \text{ and } \mathbf{P}\mathbf{X}_1 = \mathbf{X}_2. \tag{7.57}$$

It is important to note that isomorphic graphs are really are identical in terms of their underlying structure. The ordering of the nodes in the adjacency matrix is an arbitrary decision we must make when we represent a graph using algebraic objects (e.g., matrices), but this ordering has no bearing on the structure of the underlying graph itself.

Despite its simple definition, testing for graph isomorphism is a fundamentally hard problem. For instance, a naive approach to test for isomorphism would involve the following optimization problem:

$$\min_{\mathbf{P}\in\mathcal{P}} \|\mathbf{P}\mathbf{A}_1\mathbf{P}^\top - \mathbf{A}_2\| + \|\mathbf{P}\mathbf{X}_1 - \mathbf{X}_2\| \stackrel{?}{=} 0. \tag{7.58}$$

This optimization requires searching over the full set of permutation matrices $\mathcal{P}$ to evaluate whether or not there exists a single permutation matrix $\mathbf{P}$ that leads to an equivalence between the two graphs. The computational complexity of this naive approach is immense at $O(|V|!)$, and in fact, no polynomial time algorithm is known to correctly test isomorphism for general graphs.

Graph isomorphism testing is formally referred to as NP-indeterminate (NPI). It is known to not be NP-complete, but no general polynomial time algorithms are known for the problem. (Integer factorization is another well-known problem that is suspected to belong to the NPI class.) There are, however, many practical algorithms for graph isomorphism testing that work on broad classes of graphs, including the WL algorithm that we introduced briefly in Chapter 1.

7.3.2 Graph Isomorphism and Representational Capacity

The theory of graph isomorphism testing is particularly useful for graph representation learning. It gives us a way to quantify the *representational power* of different learning approaches. If we have an algorithm—for example, a GNN—that can generate representations $\mathbf{z}_\mathcal{G} \in \mathbb{R}^d$ for graphs, then we can quantify the power of this learning algorithm by asking how useful these representations would be for testing graph isomorphism. In particular, given learned representations $\mathbf{z}_{\mathcal{G}_1}$ and $\mathbf{z}_{\mathcal{G}_2}$ for two graphs, a "perfect" learning algorithm would have that

$$\mathbf{z}_{\mathcal{G}_1} = \mathbf{z}_{\mathcal{G}_2} \text{ if and only if } \mathcal{G}_1 \text{ is isomorphic to } \mathcal{G}_2. \tag{7.59}$$

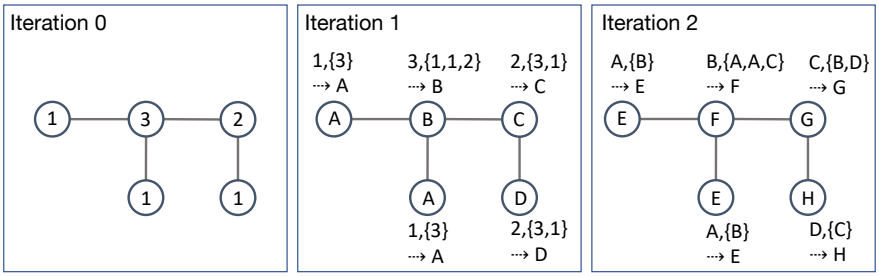

Figure 7.2: Example of the WL iterative labeling procedure on one graph.

A perfect learning algorithm would generate identical embeddings for two graphs if and only if those two graphs were actually isomorphic.

Of course, in practice, no representation learning algorithm is going to be "perfect" (unless P=NP). Nonetheless, quantifying the power of a representation learning algorithm by connecting it to graph isomorphism testing is very useful. Despite the fact that graph isomorphism testing is not solvable in general, we do know several powerful and well-understood approaches for approximate isomorphism testing, and we can gain insight into the power of GNNs by comparing them to these approaches.

7.3.3 The Weisfieler-Lehman Algorithm

The most natural way to connect GNNs to graph isomorphism testing is based on connections to the family of Weisfieler-Lehman (WL) algorithms. In Chapter 1, we discussed the WL algorithm in the context of graph kernels. However, the WL approach is more broadly known as one of the most successful and well-understood frameworks for approximate isomorphism testing. The simplest version of the WL algorithm—commonly known as the 1-WL—consists of the following steps:

1. Given two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ we assign an initial label $l^{(0)}_{\mathcal{G}_i}(v)$ to each node in each graph. In most graphs, this label is simply the node degree, i.e., $l^{(0)}(v) = d_v \, \forall v \in V$, but if we have discrete features (i.e., one hot features $\mathbf{x}_v$) associated with the nodes, then we can use these features to define the initial labels.

2. Next, we iteratively assign a new label to each node in each graph by hashing the multi-set of the current labels within the node's neighborhood, as well as the node's current label:

$$l^{(i)}_{\mathcal{G}_i}(v) = \text{HASH}(l^{(i-1)}_{\mathcal{G}_i}(v), \{\{l^{(i-1)}_{\mathcal{G}_i}(u) \, \forall u \in \mathcal{N}(v)\}\}), \tag{7.60}$$

where the double-braces are used to denote a multi-set and the HASH function maps each unique multi-set to a unique new label.

3. We repeat Step 2 until the labels for all nodes in both graphs converge, i.e., until we reach an iteration K where $l^{(K)}_{\mathcal{G}_j}(v) = l^{(K-1)}_{\mathcal{G}_i}(v), \forall v \in V_j, j = 1, 2$.

4. Finally, we construct multi-sets

$$L_{\mathcal{G}_j} = \{\{l^{(i)}_{\mathcal{G}_j}(v), \forall v \in \mathcal{V}_j, i = 0, ..., K-1\}\}$$

summarizing all the node labels in each graph, and we declare $\mathcal{G}_1$ and $\mathcal{G}_2$ to be isomorphic if and only if the multi-sets for both graphs are identical, i.e., if and only if $L_{\mathcal{G}_1} = L_{\mathcal{G}_2}$.

Figure 7.2 illustrates an example of the WL labeling process on one graph. At each iteration, every node collects the multi-set of labels in its local neighborhood, and updates its own label based on this multi-set. After K iterations of this labeling process, every node has a label that summarizes the structure of its K-hop neighborhood, and the collection of these labels can be used to characterize the structure of an entire graph or subgraph.

The WL algorithm is known to converge in at most $|\mathcal{V}|$ iterations and is known to known to successfully test isomorphism for a broad class of graphs [Babai and Kucera, 1979]. There are, however, well known cases where the test fails, such as the simple example illustrated in Figure 7.3.

Figure 7.3: Example of two graphs that cannot be distinguished by the basic WL algorithm.

7.3.4 GNNs and the WL Algorithm

There are clear analogies between the WL algorithm and the neural message passing GNN approach. In both approaches, we iteratively aggregate information from local node neighborhoods and use this aggregated information to update the representation of each node. The key distinction between the two approaches is that the WL algorithm aggregates and updates discrete labels (using a hash function) while GNN models aggregate and update node embeddings using neural networks. In fact, GNNs have been motivated and derived as a continuous and differentiable analog of the WL algorithm.

The relationship between GNNs and the WL algorithm (described in Section 7.3.3) can be formalized in the following theorem:

Theorem 4 ([Morris et al., 2019, Xu et al., 2019])**.** *Define a message-passing GNN (MP-GNN) to be any GNN that consists of K message-passing layers of the following form:*

$$\mathbf{h}_u^{(k+1)} = \textsc{update}^{(k)}\left(\mathbf{h}_u^{(k)}, \textsc{aggregate}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right), \tag{7.61}$$

where AGGREGATE *is a differentiable permutation invariant function and* UPDATE *is a differentiable function. Further, suppose that we have only discrete feature inputs at the initial layer, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u \in \mathbb{Z}^d, \forall u \in \mathcal{V}$. Then we have that $\mathbf{h}_u^{(K)} \neq \mathbf{h}_v^{(K)}$ only if the nodes u and v have different labels after K iterations of the WL algorithm.*

In intuitive terms, Theorem 7.3.4 states that GNNs are *no more powerful than the WL algorithm* when we have discrete information as node features. If the WL algorithm assigns the same label to two nodes, then any message-passing GNN will also assign the same embedding to these two nodes. This result on node labeling also extends to isomorphism testing. If the WL test cannot distinguish between two graphs, then a MP-GNN is also incapable of distinguishing between these two graphs. We can also show a more positive result in the other direction:

Theorem 5 ([Morris et al., 2019, Xu et al., 2019])**.** *There exists a MP-GNN such that $\mathbf{h}_u^{(K)} = \mathbf{h}_v^{(K)}$ if and only if the two nodes u and v have the same label after K iterations of the WL algorithm.*

This theorem states that there exist message-passing GNNs that are as powerful as the WL test.

Which MP-GNNs are most powerful? The two theorems above state that message-passing GNNs are at most as powerful as the WL algorithm and that there exist message-passing GNNs that are as powerful as the WL algorithm. So which GNNs actually obtain this theoretical upper bound? Interestingly, the basic GNN that we introduced at the beginning of Chapter 5 is sufficient to satisfy this theory. In particular, if we define the message passing updates as follows:

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)}\sum_{v \in \mathcal{N}(u)}\mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right), \tag{7.62}$$

then this GNN is sufficient to match the power of the WL algorithm [Morris et al., 2019].

However, most of the other GNN models discussed in Chapter 5 are *not* as powerful as the WL algorithm. Formally, to be as powerful as the WL algorithm, the AGGREGATE and UPDATE functions need to be *injective* [Xu et al., 2019]. This means that the AGGREGATE and UPDATE operators need to be map every unique input to a unique output value, which is not the case for many of the models we discussed. For example, AGGREGATE functions that use a (weighted) average of the neighbor embeddings are not

injective; if all the neighbors have the same embedding then a (weighted) average will not be able to distinguish between input sets of different sizes.

Xu et al. [2019] provide a detailed discussion of the relative power of various GNN architectures. They also define a "minimal" GNN model, which has few parameters but is still as powerful as the WL algorithm. They term this model the *Graph Isomorphism Network (GIN)*, and it is defined by the following update:

$$\mathbf{h}_u^{(k)} = \mathbf{MLP}^{(k)}\left((1+\epsilon^{(k)})\mathbf{h}_u^{(k-1)} + \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(k-1)}\right), \tag{7.63}$$

where $\epsilon^{(k)}$ is a trainable parameter.

7.3.5 Beyond the WL Algorithm

The previous subsection highlighted an important negative result regarding message-passing GNNs (MP-GNNs): these models are no more powerful than the WL algorithm. However, despite this negative result, investigating how we can make GNNs that are provably *more powerful* than the WL algorithm is an active area of research.

Relational pooling

One way to motivate provably more powerful GNNs is by considering the failure cases of the WL algorithm. For example, we can see in Figure 7.3 that the WL algorithm—and thus all MP-GNNs—cannot distinguish between a connected 6-cycle and a set of two triangles. From the perspective of message passing, this limitation stems from the fact that AGGREGATE and UPDATE operations are unable to detect when two nodes share a neighbor. In the example in Figure 7.3, each node can infer from the message passing operations that they have two degree-2 neighbors, but this information is not sufficient to detect whether a node's neighbors are connected to one another. This limitation is not simply a corner case illustrated in Figure 7.3. Message passing approaches generally fail to identify closed triangles in a graph, which is a critical limitation.

To address this limitation, Murphy et al. [2019] consider augmenting MP-GNNs with *unique node ID features*. If we use MP-GNN$(\mathbf{A}, \mathbf{X})$ to denote an arbitrary MP-GNN on input adjacency matrix $\mathbf{A}$ and node features $\mathbf{X}$, then adding node IDs is equivalent to modifying the MP-GNN to the following:

$$\text{MP-GNN}(\mathbf{A}, \mathbf{X} \oplus \mathbf{I}), \tag{7.64}$$

where $\mathbf{I}$ is the $d \times d$-dimensional identity matrix and $\oplus$ denotes column-wise matrix concatenation. In other words, we simply add a unique, one-hot indicator feature for each node. In the case of Figure 7.3, adding unique node IDs would

allow a MP-GNN to identify when two nodes share a neighbor, which would make the two graphs distinguishable.

Unfortunately, however, this idea of adding node IDs does not solve the problem. In fact, by adding unique node IDs we have actually introduced a new and equally problematic issue: the MP-GNN is no longer permutation equivariant. For a standard MP-GNN we have that

$$\mathbf{P}(\text{MP-GNN}(\mathbf{A}, \mathbf{X})) = \text{MP-GNN}(\mathbf{PAP}^\top, \mathbf{PX}), \tag{7.65}$$

where $\mathbf{P} \in \mathcal{P}$ is an arbitrary permutation matrix. This means that standard MP-GNNs are permutation equivariant. If we permute the adjacency matrix and node features, then the resulting node embeddings are simply permuted in an equivalent way. However, MP-GNNs with node IDs are not permutation invariant since in general

$$\mathbf{P}(\text{MP-GNN}(\mathbf{A}, \mathbf{X} \oplus \mathbf{I})) \neq \text{MP-GNN}(\mathbf{PAP}^\top, (\mathbf{PX}) \oplus \mathbf{I}). \tag{7.66}$$

The key issue is that assigning a unique ID to each node fixes a particular node ordering for the graph, which breaks the permutation equivariance.

To alleviate this issue, Murphy et al. [2019] propose the *Relational Pooling (RP)* approach, which involves marginalizing over all possible node permutations. Given any MP-GNN the RP extension of this GNN is given by

$$\text{RP-GNN}(\mathbf{A}, \mathbf{X}) = \sum_{\mathbf{P} \in \mathcal{P}} \text{MP-GNN}(\mathbf{PAP}^\top, (\mathbf{PX}) \oplus \mathbf{I}). \tag{7.67}$$

Summing over all possible permutation matrices $\mathbf{P} \in \mathcal{P}$ recovers the permutation invariance, and we retain the extra representational power of adding unique node IDs. In fact, Murphy et al. [2019] prove that the RP extension of a MP-GNN can distinguish graphs that are indistinguishable by the WL algorithm.

The limitation of the RP approach is in its computational complexity. Naively evaluating Equation (7.67) has a time complexity of $O(|\mathcal{V}|!)$, which is infeasible in practice. Despite this limitation, however, Murphy et al. [2019] show that the RP approach can achieve strong results using various approximations to decrease the computation cost (e.g., sampling a subset of permutations).

The k-WL test and k-GNNs

The Relational Pooling (RP) approach discussed above can produce GNN models that are provably more powerful than the WL algorithm. However, the RP approach has two key limitations:

1. The full algorithm is computationally intractable.
2. We know that RP-GNNs are more powerful than the WL test, but we have no way to characterize *how much more* powerful they are.

To address these limitations, several approaches have considered improving GNNs by adapting generalizations of the WL algorithm.

The WL algorithm we introduced in Section 7.3.3 is in fact just the simplest of what is known as the *family of k-WL algorithms*. In fact, the WL algorithm we introduced previously is often referred to as the *1-WL algorithm*, and it can be generalized to the *k-WL algorithm* for $k > 1$. The key idea behind the k-WL algorithms is that we label subgraphs of size k rather than individual nodes. The k-WL algorithm generates representation of a graph $\mathcal{G}$ through the following steps:

1. Let $s = (u_1, u_2, ..., u_k) \in \mathcal{V}^k$ be a tuple defining a subgraph of size k, where $u_1 \neq u_2 \neq ... \neq u_k$. Define the initial label $l_{\mathcal{G}}^{(0)}(s)$ for each subgraph by the isomorphism class of this subgraph (i.e., two subgraphs get the same label if and only if they are isomorphic).

2. Next, we iteratively assign a new label to each subgraph by hashing the multi-set of the current labels within this subgraph's neighborhood:

$$l_{\mathcal{G}}^{(i)}(s) = \text{HASH}(\{\{l_{\mathcal{G}}^{(i-1)}(s'), \forall s' \in \mathcal{N}_j(s), j = 1, ..., k\}\}, l_{\mathcal{G}}^{(i-1)}(s)),$$

   where the jth subgraph neighborhood is defined as

$$\mathcal{N}_j(s) = \{\{(u_1, ..., u_{j-1}, v, u_{j+1}, ..., u_k), \forall v \in \mathcal{V}\}\}. \tag{7.68}$$

3. We repeat Step 2 until the labels for all subgraphs converge, i.e., until we reach an iteration K where $l_{\mathcal{G}}^{(K)}(s) = l_{\mathcal{G}}^{(K-1)}(s)$ for every k-tuple of nodes $s \in \mathcal{V}^k$.

4. Finally, we construct a multi-set

$$L_{\mathcal{G}} = \{\{l_{\mathcal{G}}^{(i)}(v), \forall s \in \mathcal{V}^k, i = 0, ..., K-1\}\}$$

   summarizing all the subgraph labels in the graph.

As with the 1-WL algorithm, the summary $L_{\mathcal{G}}$ multi-set generated by the k-WL algorithm can be used to test graph isomorphism by comparing the multi-sets for two graphs. There are also graph kernel methods based on the k-WL test [Morris et al., 2019], which are analogous to the WL-kernel introduced that was in Chapter 1.

An important fact about the k-WL algorithm is that it introduces a hierarchy of representational capacity. For any $k \geq 2$ we have that the $(k+1)$-WL test is strictly more powerful than the k-WL test.[6] Thus, a natural question to ask is whether we can design GNNs that are as powerful as the k-WL test for $k > 2$, and, of course, a natural design principle would be to design GNNs by analogy to the k-WL algorithm.

Morris et al. [2019] attempt exactly this: they develop a k-GNN approach that is a differentiable and continuous analog of the k-WL algorithm. k-GNNs

[6]However, note that running the k-WL requires solving graph isomorphism for graphs of size k, since Step 1 in the k-WL algorithm requires labeling graphs according to their isomorphism type. Thus, running the k-WL for $k > 3$ is generally computationally intractable.

learn embeddings associated with subgraphs—rather than nodes—and the message passing occurs according to subgraph neighborhoods (e.g., as defined in Equation 7.68). Morris et al. [2019] prove that k-GNNs can be as expressive as the k-WL algorithm. However, there are also serious computational concerns for both the k-WL test and k-GNNs, as the time complexity of the message passing explodes combinatorially as k increases. These computational concerns necessitate various approximations to make k-GNNs tractable in practice [Morris et al., 2019].

Invariant and equivariant k-order GNNs

Another line of work that is motivated by the idea of building GNNs that are as powerful as the k-WL test are the invariant and equivariant GNNs proposed by Maron et al. [2019]. A crucial aspect of message-passing GNNs (MP-GNNs; as defined in Theorem 7.3.4) is that they are equivariant to node permutations, meaning that

$$\mathbf{P}(\text{MP-GNN}(\mathbf{A}, \mathbf{X})) = \text{MP-GNN}(\mathbf{PAP}^\top, \mathbf{PX}). \tag{7.69}$$

for any permutation matrix $\mathbf{P} \in \mathcal{P}$. This equality says that that permuting the input to an MP-GNN simply results in the matrix of output node embeddings being permuted in an analogous way.

In addition to this notion of equivariance, we can also define a similar notion of permutation *invariance* for MP-GNNs at the graph level. In particular, MP-GNNs can be extended with a $\text{POOL} : \mathbb{R}^{|\mathcal{V}| \times d} \to \mathbb{R}$ function (see Chapter 5), which maps the matrix of learned node embeddings $\mathbf{Z} \in \mathbb{R}^{|\mathcal{V}| \times d}$ to an embedding $\mathbf{z}_\mathcal{G} \in \mathbb{R}^d$ of the entire graph. In this graph-level setting we have that MP-GNNs are permutation invariant, i.e.

$$\text{POOL}\left(\text{MP-GNN}(\mathbf{PAP}^\top, \mathbf{PX})\right) = \text{POOL}\left(\text{MP-GNN}(\mathbf{A}, \mathbf{X})\right), \tag{7.70}$$

meaning that the pooled graph-level embedding does not change when different node orderings are used.

Based on this idea of invariance and equivariance, Maron et al. [2019] propose a general form of GNN-like models based on permutation equivariant/invariant tensor operations. Suppose we have an order-$(k+1)$ tensor $\mathcal{X} \in \mathbb{R}^{|\mathcal{V}|^k \times d}$, where we assume that the first k channels/modes of this tensor are indexed by the nodes of the graph. We use the notation $\mathbf{P} \star \mathcal{X}$ to denote the operation where we permute the first k channels of this tensor according the node permutation matrix $\mathbf{P}$. We can then define an linear equivariant layer as a linear operator (i.e., a tensor) $\mathcal{L} : \mathbb{R}^{|\mathcal{V}|^{k_1} \times d_1} \to \mathbb{R}^{|\mathcal{V}|^{k_2} \times d_2}$:

$$\mathcal{L} \times (\mathbf{P} \star \mathcal{X}) = \mathbf{P} \star (\mathcal{L} \times \mathcal{X}), \forall \mathbf{P} \in \mathcal{P}, \tag{7.71}$$

where we use $\times$ to denote a generalized tensor product. Invariant linear operators can be similarly defined as tensors $\mathcal{L}$ that satisfy the following equality:

$$\mathcal{L} \times (\mathbf{P} \star \mathcal{X}) = \mathcal{L} \times \mathcal{X}, \forall \mathbf{P} \in \mathcal{P}. \tag{7.72}$$

Note that both equivariant and invariant linear operators can be represented as tensors, but they have different structure. In particular, an equivariant operator $\mathcal{L} : \mathbb{R}^{|\mathcal{V}|^k \times d_1} \rightarrow \mathbb{R}^{|\mathcal{V}|^k \times d_2}$ corresponds to a tensor $\mathcal{L} \in \mathbb{R}^{|\mathcal{V}|^{2k} \times d_1 \times d_2}$, which has $2k$ channels indexed by nodes (i.e., twice as many node channels as the input). On the other hand, an invariant operator $\mathcal{L} : \mathbb{R}^{|\mathcal{V}|^k \times d_1} \rightarrow \mathbb{R}^{d_2}$ corresponds to a tensor $\mathcal{L} \in \mathbb{R}^{|\mathcal{V}|^k \times d_1 \times d_2}$, which has k channels indexed by nodes (i.e., the same number as the input). Interestingly, taking this tensor view of the linear operators, the equivariant (Equation 7.71) and invariant (Equation 7.72) properties for can be combined into a single requirement that the $\mathcal{L}$ tensor is a fixed point under node permutations:

$$\mathbf{P} \star \mathcal{L} = \mathcal{L}, \forall \mathbf{P} \in \mathcal{P}. \tag{7.73}$$

In other words, for a given input $\mathcal{X} \in \mathbb{R}^{|V|^k \times d}$, both equivariant and invariant linear operators on this input will correspond to tensors that satisfy the fixed point in Equation (7.73), but the number of channels in the tensor will differ depending on whether it is an equivariant or invariant operator.

Maron et al. [2019] show that tensors satisfying the the fixed point in Equation (7.73) can be constructed as a linear combination of a set of fixed basis elements. In particular, any order-l tensor $\mathcal{L}$ that satisfies Equation (7.73) can be written as

$$\mathcal{L} = \beta_1 \mathcal{B}_1 + \beta_2 + ... + \beta_{b(l)} \mathcal{B}_{b(l)}, \tag{7.74}$$

where $\mathcal{B}_i$ are a set of fixed basis tensors, $\beta_i \in \mathbb{R}$ are real-valued weights, and $b(l)$ is the lth *Bell number*. The construction and derivation of these basis tensors is mathematically involved and is closely related to the theory of Bell numbers from combinatorics. However, a key fact and challenge is that the number of basis tensors needed grows with lth Bell number, which is an exponentially increasing series.

Using these linear equivariant and invariant layers, Maron et al. [2019] define their invariant k-order GNN model based on the following composition of functions:

$$\mathbf{MLP} \circ \mathcal{L}_0 \circ \sigma \circ \mathcal{L}_1 \circ \sigma \mathcal{L}_2 \cdots \sigma \circ \mathcal{L}_m \times \mathcal{X}. \tag{7.75}$$

In this composition, we apply m equivariant linear layers $\mathcal{L}_1, ..., \mathcal{L}_m$, where $L_i : \mathcal{L} : \mathbb{R}^{|\mathcal{V}|^{k_i} \times d_1} \rightarrow \mathbb{R}^{|\mathcal{V}|^{k_{i+1}} \times d_2}$ with $\max_i k_i = k$ and $k_1 = 2$. Between each of these linear equivariant layers an element-wise non-linearity, denoted by σ, is applied. The penultimate function in the composition, is an invariant linear layer, $\mathcal{L}_0$, which is followed by a multi-layer perceptron (MLP) as the final function in the composition. The input to the k-order invariant GNN is the tensor $\mathcal{X} \in \mathbb{R}^{|\mathcal{V}|^2 \times d}$, where the first two channels correspond to the adjacency matrix and the remaining channels encode the initial node features/labels.

This approach is called k-order because the equivariant linear layers involve tensors that have up to k different channels. Most importantly, however, Maron et al. [2019] prove that k-order models following Equation 7.75 are equally powerful as the k-WL algorithm. As with the k-GNNs discussed in the previous section, however, constructing k-order invariant models for $k > 3$ is generally computationally intractable.