https://doi.org/10.1142/9789811246081_0014

Chapter 14

Deep Learning for Graph-Structured Data

Luca Pasa, Nicolò Navarin†, and Alessandro Sperduti‡*

Department of Mathematics, University of Padua
Via Trieste, 63 - 35121 Padova, Italy
**luca.pasa@unipd.it*
†*nicolo.navarin@unipd.it*
‡*alessandro.sperduti@unipd.it*

In this chapter, we discuss the application of deep learning techniques to input data that exhibit a graph structure. We consider both the case in which the input is a single, huge graph (e.g., a social network), where we are interested in predicting the properties of single nodes (e.g., users), and the case in which the dataset is composed of many small graphs where we want to predict the properties of whole graphs (e.g., molecule property prediction). We discuss the main components required to define such neural architectures and their alternative definitions in the literature. Finally, we present experimental results comparing the main graph neural networks in the literature.

14.1. Introduction

In this chapter, we consider whether it is possible to define neural network architectures that deal with structured input data. The amount and variety of data generated and stored in modern information processing systems is constantly increasing, as is the use of machine learning (ML) approaches, especially very successful deep learning (DL) solutions, to extract knowledge from them. Traditional ML approaches have been developed assuming data to be encoded into feature vectors; however, many important real-world applications generate data that are naturally represented by more complex structures, such as graphs. Graphs are particularly suited to represent the relations (arcs) between the components (nodes) constituting an entity. For instance, in social network data, single data "points" (i.e., users) are closely inter-related, and not explicitly representing such dependencies would most likely

lead to an information loss. Because of this, ML techniques able to directly process structured data have gained more and more attention since the first developments, such as recursive neural networks [1, 2] proposed in the second half of the 1990s. In the last few years, there has been a burst of interest in developing DL models for graph domains, or more in general, for non-Euclidean domains, commonly referred to as *geometric deep learning* [3]. In this chapter, we focus on deep learning for graphs.

14.1.1. *Why Graph Neural Networks?*

The analysis of structured data is not new to the machine learning community. After the first proposals in the field of neural networks [1, 2, 4], in the 2000s, kernel methods for structured data became the dominant approach to dealing with such kinds of data. Kernel methods have been defined for trees [5], graphs, [6–8] and graph with continuous attributes [9, 10]. Kernel methods have been quite successful on the many tasks where small- to medium-sized datasets were available. However, the kernel approach suffers from two main drawbacks:

- lack of scalability to large datasets (with some exceptions);
- scarce ability to deal with real-valued attributes.

For more details on kernel methods for structured data, please refer to the surveys by Giannis et al. [11] or Nils et al. [12]. In the beginning of the 2010s, following the success of deep neural networks in many application domains, in particular of convolutional networks for images [13], the research community started to be interested in the design of deep neural network models for graphs, hoping for a leap in performance, just as it happened in the case of images. The first scientific papers in this research area laid down the basic concepts that are still in use. The core idea is to learn a representation of a node in the graph that is conditioned on the representations of neighboring nodes. While this approach can be easily implemented in trees or DAGs, where the presence of directed arcs allows the updating of the nodes in a reverse topological order, in general graphs the presence of cycles is an obstacle to its implementation. A practical solution that has been identified by earlier researchers [14, 15] consists in using an iterative or multilayer approach, in which each node representation depends on the ones of the neighbors *at the previous iteration or layer*. A few years later, basically the same idea has been derived and implemented, starting from spectral graph theory (see Section 14.4). Having the possibility to learn a representation of a node in a graph is functional to the design of predictors in two main settings, namely prediction of a property of a single node in a (large) graph or of a property of a graph as a whole.

14.1.2. *The Two Main Settings: Node Classification vs. Graph Classification*

There are two main problem settings that can arise when dealing with structured data.

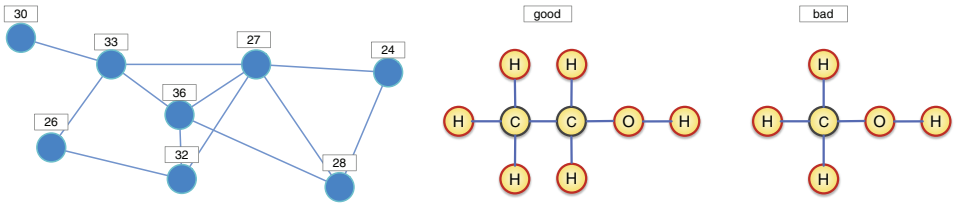
- **Predictions over nodes in a network.** In this setting, the dataset is composed of a single (possibly disconnected) large graph. Each example is a node in the graph, and the learning tasks are defined as predictions over the nodes. An example in this setting is the prediction of properties of a social network user based on his or her connections.
- **Predictions over graphs.** In this case, each example is composed of a whole graph, and the learning tasks are predictions of properties of the whole graphs. An example is the prediction of toxicity in humans of chemical compounds represented by their molecular graph.

Figure 14.1 shows the difference between the two settings. From a technical point of view, both settings require us to define (deep) neural architectures able to learn *graph isomorphic invariant representations* for nodes, which is usually obtained by resorting to a *graph convolution*. However, the second setting requires additional components, as will be discussed in Section 14.7, for learning a graph isomorphic invariant representation for each individual graph.

14.1.3. *Graph Neural Networks Basics*

As mentioned before, the main problem faced by graph neural networks is to compute a *sound* and *meaningful* representation for graph nodes, i.e.,

- it is invariant to the way the graph is represented (e.g., in which order nodes and arcs of a graph are presented);
- it incorporates structural information that is relevant for the prediction task.



(a) Node-level predictions. (b) Graph-level predictions.

Figure 14.1: Examples of tasks on graphs: (a) node regression task, e.g., prediction of the age of users in a social network; (b) graph classification task, e.g., molecule classification. The values inside the boxes are the targets we aim to predict. Note that in (a) there is a target for each node, while in (b) we have a single target per graph.

The first requirement (*soundness*) is justified by the fact that graphs that are *isomorphic* should induce the same function over their nodes. This requirement is so important that even the representations of whole graphs should comply to it; otherwise the same graph represented in different ways may return different predictions, which is of course not desirable. The second requirement (*meaningfulness*) is obvious: learned representations should preserve only the necessary information to best perform the task at hand.

Both requirements can be reasonably satisfied in a neural network by using a convolution operator, defined on graphs, supported by a mechanism of communication (message-passing) between neighboring nodes.

The general idea of graph convolution starts from a parallel between graphs and images. In convolutional networks for images, the hidden representation inherits the shape of the input image (with the exception of pixels close to the border that, if no padding is considered, imply a slight dimension reduction). Let us focus on the first convolutional layer. For each entry in the hidden representation, the convolution layer computes a representation that depends on the corresponding input pixel, and on the neighboring ones (depending on the size of the filter). In practice, we have that each neighboring position has an associated weight. This definition of convolution layer stems from the direct application of the convolution operator.

On graphs, there is not a straightforward corresponding convolution operator. As we will see later, it is possible either to define graph neural networks directly in the graph domain (see Section 14.3), or to resort to the graph spectral domain (see Section 14.4).

14.2. Notation and Definitions

In the following, we use italic letters to refer to variables, bold lowercase letters to refer to vectors, and bold uppercase letters to refer to matrices. The elements of a matrix $\mathbf{A}$ are referred to as a_{ij} (and similarly for vectors). We use uppercase letters to refer to sets or tuples.

In this chapter, we deal with problems and tasks that involve the concept of graph. Let $G = (V, E, \mathbf{X})$ be a graph, where $V = \{v_0, \ldots, v_{n-1}\}$ denotes the set of vertices (or nodes) of the graph, $E \subseteq V \times V$ is the set of edges, and $\mathbf{X} \in \mathbb{R}^{n \times s}$ is a multivariate signal on the graph nodes with the i-th row $\mathbf{x}_i$ representing the attributes of v_i. We define $\mathbf{A} \in \mathbb{R}^{n \times n}$ as the adjacency matrix of the graph, with elements $a_{ij} = 1 \iff (v_i, v_j) \in E$. With $\mathcal{N}(v)$ we denote the set of nodes adjacent to node v.

14.3. Early Models of Graph Neural Network

In this section, we review the early definitions of the graph neural networks that have been proposed in literature. In the following definitions, for the sake of simplicity,

we ignore the bias terms. During the reading of this section, it is important to keep in mind that a graph convolution operator can be applied to perform both graph classification and node classification (see Section 14.7). Therefore, while some of them have been applied to graph classification rather than node classification, the basic node relabeling or graph convolution components are the same for both tasks.

The first definition of neural network for structured data, including graphs, has been proposed by Sperduti and Starita in 1997 [2]. The authors propose to use the generalized recursive neuron, a generalization to structures of a recurrent neuron, which is able to build a map from a domain of structures to the set of reals. Using the generalized recursive neurons, the authors show that it is possible to formalize several learning algorithms and models able to deal with structures as generalization of the supervised networks developed for sequences. The core idea is to define a neural architecture that is modeled according to the graph topology. Thanks to weights sharing, the same set of neurons is applied to each vertex in the graph, obtaining a representation in output that is based on the information associated with the vertex and on the representations generated for its neighbors. Although in the paper a learning algorithm based on *recurrent backpropagation* [16, 17] is proposed for general graphs with cycles, no experimental assessment of the algorithm is reported. Later, an approach for general graphs, based on standard backpropagation, has been proposed by Micheli [15] for graph predictions, and by Scarselli et al. [14] for node prediction. Several years later, the approach followed by Micheli was independently (re)proposed in [18] under the name *graph convolution*.

14.3.1. *Recursive Graph Neural Networks*

Scarselli et al. [14] proposed a recursive neural network exploiting the following general form of convolution:

$$\mathbf{h}_v^{t+1} = \sum_{u \in \mathcal{N}(v)} f(\mathbf{h}_u^t, \mathbf{x}_v, \mathbf{x}_u), \tag{14.1}$$

where $t \geq 0$ is the time of the recurrence, $\mathbf{h}_v^{t+1}$ is the representation of node v at timestep $t+1$, and f is a neural network whose parameters have to be learned, and are shared among all the vertices. The recurrent system is defined as a contraction mapping, and thus it is guaranteed to converge to a fixed point $\mathbf{h}^\star$. It can be noticed that $\mathbf{h}^\star$ is *sound* since: (i) the terms $f(\mathbf{h}_u^t, \mathbf{x}_v, \mathbf{x}_u)$ do not change with a change in the graph representation; (ii) the terms just mentioned are aggregated by a commutative operator (i.e., sum), and so $\mathbf{h}_v^{t+1}$ ($t \geq 0$) does not depend on the order of presentation of the vertices. This idea has been re-branded later as *neural message passing*, the formal definition of which was proposed by Gilmer et al. [19]

14.3.2. *Feedforward Graph Neural Networks*

In the same year, Micheli [15] proposed the neural network for graphs (NN4G) model. NN4G is based on a graph convolution that is defined as:

$$\mathbf{h}_v^{(1)} = \sigma\left(\bar{\mathbf{W}}^{(1)}\mathbf{x}_v\right), \tag{14.2}$$

$$\mathbf{h}_v^{(i+1)} = \sigma\left(\bar{\mathbf{W}}^{(i+1)}\mathbf{x}_v + \sum_{k=1}^{i}\hat{\mathbf{W}}^{(i+1,k)}\sum_{u\in\mathcal{N}(v)}\mathbf{h}_u^{(k)}\right), \quad i > 0, \tag{14.3}$$

where $1 \leq v \leq n$ is the vertex index, $\hat{\mathbf{W}}^{(l,m)}$ are the weights on the connections from layer m to layer l, $\bar{\mathbf{W}}^{(i+1)}$ are weights transforming the input representations for the $(i+1)$-th layer, and σ is a nonlinear activation function applied element-wise. Note that in this formulation, skip connections are present, to the $(i+1)$-th layer, from layer 1 to layer i. Moreover, *soundness* of $\mathbf{h}_v^{(i+1)}$ for $i \geq 1$ is again guaranteed by the commutativity of the sum over the neighbours (i.e., $\sum_{u\in\mathcal{N}(v)}\mathbf{h}_u^{(k)}$).

14.4. Derivation of Spectral Graph Convolutions

The derivation of the spectral graph convolution operator originates from graph spectral filtering [18]. Let us fix a graph G. Let $x : V \to \mathbb{R}$ be a signal on the nodes V of the graph G, i.e., a function that associates a real value with each node of V. Since the number of nodes in G is fixed (i.e., n) and the set V can be arbitrarily but consistently ordered[1], we can naturally represent every signal as a vector $\mathbf{x} \in \mathbb{R}^n$, which from now on we will refer to as signal. In order to set up a convolutional network on G, we need the notion of convolution $*_G$ between a signal $\mathbf{x}$ and a filter signal $\mathbf{f}$. However, as we do not have a natural description of translation on graphs, it is not so obvious how to define the convolution directly in the graph domain. This operation is therefore usually defined in the spectral domain of the graph, using an analogy with classical Fourier analysis in which the convolution of two signals is computed as the pointwise product of their Fourier transforms.

For this reason, we start providing the definition of the graph Fourier transform [20]. Let $\mathbf{L}$ be the (normalized) graph Laplacian, defined as

$$\mathbf{L} = \mathbf{I_n} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}, \tag{14.4}$$

[1]The set V is organized in an arbitrary order, for instance, when representing the graph with the corresponding adjacency matrix.

where $\mathbf{I_n}$ is the $n \times n$ identity matrix, and $\mathbf{D}$ is the degree matrix with entries given as

$$d_{ij} = \begin{cases} \sum_{k=0}^{n-1} a_{ik}, & \text{if } i = j \\ 0, & \text{otherwise} \end{cases}. \tag{14.5}$$

Since $\mathbf{L}$ is real, symmetric, and positive semi-definite, we can compute its eigendecomposition as

$$\mathbf{L} = \mathbf{U}\Lambda\mathbf{U}^\top, \tag{14.6}$$

where $\Lambda = \text{diag}(\lambda_0, \ldots, \lambda_{n-1})$ is a diagonal matrix with the ordered eigenvalues of $\mathbf{L}$ as diagonal entries, and the orthonormal matrix $\mathbf{U}$ contains the corresponding eigenvectors $\{\mathbf{u}_0, \ldots, \mathbf{u}_{n-1}\}$ of $\mathbf{L}$ as columns. In many classical settings of Fourier analysis, such as the Euclidean space or the torus, the Fourier transform can be defined in terms of the eigenvalues and eigenvectors of the Laplace operator. In analogy, we consider now the eigenvectors $\{\mathbf{u}_0, \ldots, \mathbf{u}_{n-1}\}$ as the Fourier basis on the graph G, and the eigenvalues $\{\lambda_0, \ldots, \lambda_{n-1}\}$ as the corresponding graph frequencies. In particular, going back to our spatial signal $\mathbf{x}$, we can define its graph Fourier transform as

$$\hat{\mathbf{x}} = \mathbf{U}^\top\mathbf{x}, \tag{14.7}$$

and its inverse graph Fourier transform as

$$\mathbf{x} = \mathbf{U}\hat{\mathbf{x}}. \tag{14.8}$$

The entries $\hat{x}_i = \mathbf{x} \cdot \mathbf{u}_i$ are the frequency components or coefficients of the signal $\mathbf{x}$ with respect with the basis function $\mathbf{u}_i$ and associated with the graph frequency λ_i. For this reason, $\hat{\mathbf{x}}$ can also be regarded as a distribution on the spectral domains of the graph, i.e., to each basis function $\mathbf{u}_i$ with frequency λ_i a corresponding coefficient $\hat{x}_i$ is associated.

Using the graph Fourier transform to switch between spatial and spectral domains, we are now ready to define the graph convolution between a filter $\mathbf{f}$ and a signal $\mathbf{x}$ as

$$\mathbf{f} *_G \mathbf{x} = \mathbf{U}\left(\hat{\mathbf{f}} \odot \hat{\mathbf{x}}\right) = \mathbf{U}\left(\left(\mathbf{U}^\top\mathbf{f}\right) \odot \left(\mathbf{U}^\top\mathbf{x}\right)\right), \tag{14.9}$$

where $\hat{\mathbf{f}} \odot \hat{\mathbf{x}} = (\hat{f}_0\hat{x}_0, \ldots, \hat{f}_{n-1}\hat{x}_{n-1})$ denotes the component-wise Hadamard product of the two vectors $\hat{\mathbf{x}}$ and $\hat{\mathbf{f}}$.

For graph convolutional networks, it is easier to design the filters $\mathbf{f}$ in the spectral domain as a distribution $\hat{\mathbf{f}}$, and then define the filter $\mathbf{f}$ on the graph as $\mathbf{f} = \mathbf{U}\hat{\mathbf{f}}$.

According to Eq. (14.9), for a given $\hat{\mathbf{f}}$ the application of the convolutional filter $\mathbf{f}$ to a signal $\mathbf{x}$ is given as

$$\mathbf{f} *_G \mathbf{x} = \mathbf{U}\left(\left(\mathbf{U}^\top \mathbf{U}\hat{\mathbf{f}}\right) \odot \left(\mathbf{U}^\top \mathbf{x}\right)\right) = \mathbf{U}\left(\hat{\mathbf{f}} \odot \left(\mathbf{U}^\top \mathbf{x}\right)\right). \tag{14.10}$$

The Hadamard product $\hat{\mathbf{f}} \odot \hat{\mathbf{x}}$ can be formulated in matrix–vector notation as $\hat{\mathbf{f}} \odot \hat{\mathbf{x}} = \hat{\mathbf{F}}\hat{\mathbf{x}}$ by applying the diagonal matrix $\hat{\mathbf{F}} = \mathrm{diag}(\hat{\mathbf{f}})$, given by

$$(\hat{\mathbf{F}})_{ij} = \left(\mathrm{diag}(\hat{\mathbf{f}})\right)_{ij} = \begin{cases} \hat{f}_i & \text{if } i = j \\ 0 & \text{otherwise} \end{cases},$$

to the vector $\hat{\mathbf{x}}$. According to Eq. (14.10), we therefore obtain

$$\mathbf{f} *_G \mathbf{x} = \mathbf{U}\hat{\mathbf{F}}\mathbf{U}^\top \mathbf{x}. \tag{14.11}$$

We can design the diagonal matrix $\hat{\mathbf{F}}$ and, thus, the spectral filter $\mathbf{f}$ in various ways. The simplest way would be to define $\mathbf{f}_\theta$ as a non-parametric filter, i.e., use $\hat{\mathbf{F}}_\theta = \mathrm{diag}(\theta)$, where $\theta = (\theta_0, \ldots, \theta_{n-1})^\top$ is a completely free vector of filter parameters that can be learned by the neural network. However, such a filter grows in size with the data, and it is not well suited for learning.

A simple alternative is to use a polynomial parametrization based on powers of the spectral matrix Λ [21] for the filter, such as,

$$\hat{\mathbf{F}}_\theta = \sum_{i=0}^{k} \theta_i \Lambda^i. \tag{14.12}$$

This filter has $k + 1$ parameters $\{\theta_0, \ldots, \theta_k\}$ to learn, and it is spatially k-localized on the graph. One of the main advantages of this filter is that we can formulate it explicitly in the graph domain. Recalling the eigendecomposition $\mathbf{L} = \mathbf{U}\Lambda\mathbf{U}^\top$ of the graph Laplacian, Eqs. (14.11) and (14.12) when combined give

$$\begin{aligned} \mathbf{f}_\theta *_G \mathbf{x} = \mathbf{U}\hat{\mathbf{F}}_\theta \mathbf{U}^\top \mathbf{x} &= \sum_{i=0}^{k} \theta_i \mathbf{U}\Lambda^i \mathbf{U}^\top \mathbf{x} \\ &= \sum_{i=0}^{k} \theta_i (\mathbf{U}\Lambda\mathbf{U}^\top)^i \mathbf{x} = \sum_{i=0}^{k} \theta_i \mathbf{L}^i \mathbf{x}. \end{aligned} \tag{14.13}$$

Note that the computation of the eigendecomposition of the graph Laplacian $\mathbf{L}$ (the cost is of the order $O(n^3)$) is feasible only for relatively small graphs (with some thousands of nodes at most). However, real-world problems involve graphs with hundreds of thousands or even millions of nodes: in these cases, the computation of the eigendecomposition of $\mathbf{L}$ is prohibitive and a filter of the form of Eq. (14.13) has clear advantages compared to a spectral filter given in the form of Eq. (14.11).

Applying the convolution defined in Eq. (14.13) to a multivariate signal $\mathbf{X} \in \mathbb{R}^{n \times s}$ and using m filters, we obtain the following definition for a single graph convolutional layer:

$$\mathbf{H} = \sum_{i=0}^{k} \mathbf{L}^i \mathbf{X} \Theta^{(i)}. \tag{14.14}$$

where $\Theta^{(i)} \in \mathbb{R}^{s \times m}$.

This layer can be directly applied in a single-layer neural network to compute the task predictions, i.e.,

$$\mathbf{Y} = \sigma \left(\sum_{i=0}^{k} \mathbf{L}^i \mathbf{X} \Theta^{(i)} \right), \tag{14.15}$$

where σ is an appropriate activation function for the output task, e.g., the *softmax* for a classification problem with m classes.

14.5. Graph Convolutions in Literature

In this section, we present in detail some of the most widely adopted graph convolutions that have been proposed in the literature.

14.5.1. *Chebyshev Graph Convolution*

The parametrization of the polynomial filter defined in Eq. (14.12) is given in the monomial basis. Alternatively, Defferrard et al. [18] proposed to use Chebyshev polynomials as a polynomial basis. In general, the usage of a Chebyshev basis improves the stability in the case of numerical approximations. By defining the operators

$$T^{(0)}(x) = 1,\ T^{(1)}(x) = x,\ \text{and for } k > 1,\ T^{(k)}(x) = 2xT^{(k-1)}(x) - T^{(k-2)}(x),$$

the filter is defined as

$$\hat{\mathbf{F}}_\theta = \sum_{i=0}^{k} \theta_i T^{(i)}(\tilde{\Lambda}), \tag{14.16}$$

where $\tilde{\Lambda} = \frac{2}{\lambda_{max}} \Lambda - \mathbf{I}_n$ is the diagonal matrix of scaled eigenvectors of the graph Laplacian. The resulting convolution is then:

$$\mathbf{f}_\theta *_G \mathbf{x} = \sum_{i=0}^{k} \theta_i T^{(i)}(\tilde{\mathbf{L}})\mathbf{x}. \tag{14.17}$$

where $\tilde{\mathbf{L}} = \frac{2}{\lambda_{max}} \mathbf{L} - \mathbf{I}_n$.

14.5.2. *Graph Convolutional Networks*

Kipf et al. [22] proposed to fix order $k = 1$ in Eq. (14.17) to obtain a linear first-order filter for each graph convolutional layer in a neural network. Additionally, they fixed $\theta_0 = \theta_1 = \theta$. They also suggested to stack these simple convolutions to obtain larger receptive fields (i.e., neighbors reachable with an increasing number of hops) in order to improve the discriminatory power of the resulting network. Specifically, the resulting convolution operator is defined as

$$\mathbf{f}_\theta *_G \mathbf{x} = \theta(\mathbf{I_n} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})\mathbf{x} = \theta(2\mathbf{I_n} - \mathbf{L})\mathbf{x}. \tag{14.18}$$

A renormalization trick to limit the eigenvalues of the resulting matrix is also introduced by the following: $\mathbf{I_n} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ is replaced by $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$, where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I_n}$ and $d_{ii} = \sum_{j=0}^{n} \tilde{a}_{ij}$. In this way, the spectral filter $\mathbf{f}_\theta$ is built not on the spectral decomposition of the graph Laplacian $\mathbf{L}$ but on the eigendecomposition of the perturbed operator $\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$.

Applying this convolution operator to a multivariate signal $\mathbf{X} \in \mathbb{R}^{n\times s}$ and using m filters, we obtain the following definition for a single graph convolutional layer:

$$\mathbf{H} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\Theta, \tag{14.19}$$

where $\Theta \in \mathbb{R}^{s\times m}$. This convolutional operation has complexity $O(|E|ms)$. To obtain a Graph Convolutional Network (GCN), several graph convolutional layers are stacked and interleaved by a nonlinear activation function, typically a ReLU.

If $\mathbf{H}^{(0)} = \mathbf{X}$, then, based on single layer convolution in Eq. (14.19), we obtain the following recursive definition for the k-th graph convolutional layer:

$$\mathbf{H}^{(k)} = ReLU(\tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{H}^{(k-1)}\Theta). \tag{14.20}$$

14.5.3. *GraphConv*

In 2019, Morris et al. [23] investigated GNNs from a theoretical point of view. Specifically, they studied the relationship between GNN and the one-dimensional *Weisfeiler–Lehman graph isomorphism heuristic* (1-WL). 1-WL is an iterative algorithm that computes a graph invariant for labeled graphs, which at each iteration produces a coloring for the nodes of the graph. The output of an iteration depends on the coloring from the previous one. At iteration 0, the algorithm uses as initial coloring the label of the node. The aim of 1-WL is to test whether two graphs $\mathcal{G}$ and $\mathcal{H}$ are isomorphic. After applying the algorithm until convergence on both graphs, if they show different color distributions, the 1-WL test concludes that the graphs are not isomorphic. If the coloring is the same, the graphs may or may not be isomorphic. In fact, this algorithm is not able to distinguish all non-isomorphic graphs (like all graph invariants), but is still a powerful heuristic, which can successfully test

isomorphism for a broad class of graphs. A more detailed introduction to 1-WL is reported in Section 14.6. As a result of this theoretical study, Morris et al. [23] defined the *GraphConv* operator inspired by the Weisfeiler–Lehman graph invariant; it is defined as follows:

$$\mathbf{H}^{(i+1)} = \mathbf{H}^{(i)}\bar{\mathbf{W}}^{(i)} + \mathbf{A}\mathbf{H}^{(i)}\hat{\mathbf{W}}^{(i)}, \tag{14.21}$$

where $\mathbf{H}^0 = \mathbf{X}$, and $\bar{\mathbf{W}}^{(i)}$ and $\hat{\mathbf{W}}^{(i)}$ are two weights matrices. It is interesting to note that this definition of convolution is very similar to NN4G (cf. Eq. (14.3)).

14.5.4. *Graph Convolutions Exploiting Personalized PageRank*

Klicpera et al. [24] proposed a graph convolution by exploiting Personalized PageRank. Let $f(\cdot)$ define a two-layer feedforward neural network. The PPNP layer is defined as

$$\mathbf{H} = \alpha\left(\mathbf{I}_n - (1-\alpha)\tilde{\mathbf{A}}\right)^{-1} f(\mathbf{X}), \tag{14.22}$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_n$. Such a filter preserves locality due to the properties of Personalized PageRank.

The same paper proposed an approximation, derived by a truncated power iteration, avoiding the expensive computation of the matrix inversion, referred to as APPNP. It is implemented as a multilayer network where the $(l+1)$-th layer is defined as

$$\mathbf{H}^{(l+1)} = (1-\alpha)\tilde{\mathbf{S}}\mathbf{H}^{(l)} + \alpha\mathbf{H}^{(0)}, \tag{14.23}$$

where $\mathbf{H}^{(0)} = f(\mathbf{X})$ and $\tilde{\mathbf{S}}$ is the renormalized adjacency matrix adopted in GCN, i.e., $\tilde{\mathbf{S}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$.

14.5.5. *GCNII*

While the GCN presented in Section 14.5.2 is very popular, one of its main drawbacks is that it is not suited to build deep networks. Actually, the authors proposed to use just two graph convolutional layers. This is a common problem of many graph convolutions, known as the over-smoothing problem [25].

In [26], an extension of GCN was proposed. Authors aimed at the definition of a graph convolution that can be adopted to build truly deep graph neural networks. To pursue this goal, they enhance the original GCN formulation in Eq. (14.20) with *Initial residual* and *Identity mapping* connections (thus the acronym GCNII). The $l+1$-th layer of GCNII is defined as

$$\mathbf{H}^{(l+1)} = \sigma\Big(\big((1-\alpha_l)\tilde{\mathbf{S}}\mathbf{H}^{(l)} + \alpha_l\mathbf{H}^{(0)}\big)\big((1-\beta_l)\mathbf{I}_n + \beta_l\mathbf{W}^{(l)}\big)\Big), \tag{14.24}$$

where α and β are hyper-parameters. Authors propose to fix $\alpha_l = 0.1$ or 0.2 for each l, and $\beta_l = \frac{\lambda}{l}$, where λ is a single hyper-parameter to tune.

14.5.6. *Graph Attention Networks*

Graph Attention Networks (GATs) [27] exploit a different convolution operator based on masked self-attention. The attention mechanism was introduced by Bahdanau et al. [28] with the goal of enhancing the performance of the encoder–decoder architecture on neural network-based machine translation tasks. The basic idea behind the proposed attention mechanism is to allow the model to selectively focus on valuable parts of the input, and hence learn the associations between them. In GAT, the attention mechanism is developed by replacing the adjacency matrix in the convolution with a matrix of attention weights

$$\mathbf{H}^{(i+1)} = \sigma(\mathbf{B}^{(i+1)}\mathbf{H}^{(i)}\Theta), \tag{14.25}$$

where $0 \leq i < l$ (the number of layers), $\mathbf{H}^{(0)} = \mathbf{X}$, and the u, v-th element of $\mathbf{B}^{(i+1)}$ is defined as

$$b_{u,v}^{(i+1)} = \frac{\exp(LeakyRELU(\mathbf{w}'^{\top}[\mathbf{W}\mathbf{h}_u^{(i)}||\mathbf{W}\mathbf{h}_v^{(i)}]))}{\sum_{k\in\mathcal{N}(u)}\exp(LeakyRELU(\mathbf{w}'^{\top}[\mathbf{W}\mathbf{h}_u^{(i)}||\mathbf{W}\mathbf{h}_k^{(i)}]))}, \tag{14.26}$$

if $(u, v) \in E$, and 0 otherwise. The vector $\mathbf{w}'$ and the matrix $\mathbf{W}$ are learnable parameters. Authors propose to use multihead attention to stabilize the training. While it may be harder to train, GAT allows to weight differently the neighbors of a node; thus it is a very expressive graph convolution.

14.5.7. *GraphSAGE*

Another interesting proposal for the convolution over the node neighborhood is GraphSage [29], where the aggregation over the neighborhoods is performed by using sum, mean or max-pooling operators, followed by a linear projection in order to update the node representation. In addition to that, the proposed approach exploits a particular neighbor sampling scheme. In fact, the method uniformly samples for each vertex v, a fixed-size set of neighbors $\mathcal{U}(v)$, instead of using the full neighborhood $\mathcal{N}(v)$. Using a fixed-size subset of neighbors allows to maintain the computational footprint of each batch invariant to the actual node degree distribution. Formally, the k-th layer of the GraphSAGE convolution is defined as follows:

$$\mathbf{h}_{\mathcal{U}(v)}^{(k)} = \mathcal{A}^{(k)}(\{\mathbf{h}_u^{(k-1)}, \forall u \in \mathcal{U}(v)\}), \tag{14.27}$$

$$\mathbf{h}_v^{(k)} = \sigma(\mathbf{W}^{(k)}[\mathbf{h}_v^{(k-1)}, \mathbf{h}_{\mathcal{U}(v)}^{(k)}]), \tag{14.28}$$

where $\mathcal{A}^{(k)}$ is the aggregation function used at the k-th layer. In particular, the authors proposed to use different strategies to implement $\mathcal{A}$, such as a mean aggregation function, a pooling aggregator, or even a more complex aggregator based on an LSTM architecture.

14.5.8. *Gated Graph Sequence Neural Networks*

Li et al. [30] extended the recurrent graph neural network proposed by Scarselli et al. in 2009 [14]. They proposed to remove the constraint for the recurrent system to be a contraction mapping, and implemented this idea by adopting recurrent neural networks to define the recurrence. Specifically, the authors adopted the gated recurrent unit (GRU). The recurrent convolution operator is defined as follows:

$$\begin{aligned}
\mathbf{h}_v^{(1)} &= [\mathbf{x}_v, \mathbf{0}], \\
\mathbf{a}_v^{(t)} &= \mathbf{A}_v[\mathbf{h}_v^{(t-1)}, \forall v \in V], \\
\mathbf{z}_v^t &= \sigma(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}), \\
\mathbf{r}_v^t &= \sigma(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}), \\
\mathbf{c}_v^t &= \tanh(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)})), \\
\mathbf{h}_v^{(t)} &= (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \mathbf{c}_v^t,
\end{aligned}$$

where $\mathbf{A}_v$ is row v of the adjacency matrix $\mathbf{A}$. Note that the propagation of the node embeddings through the edges is performed by computing $\mathbf{a}_v^{(t)}$, while the other equations describe the GRU unit structure.

14.5.9. *Other Convolutions*

The convolutions presented up to now are the most common and interesting. Moreover, the structure of these operators highlights many elements that the majority of the GCs developed in the last few years have in common. In the literature, many convolutions were developed using some of the discussed operators as a starting point. For instance, DGCNN [31] adopts a graph convolution very similar to GCN [22]. Specifically, it adopts a slightly different propagation scheme for vertices' representations, based on the random-walk graph Laplacian.

A more straightforward approach in defining convolutions on graphs is PATCHY-SAN (PSCN) [32]. This approach is inspired by how convolutions are defined over images. It consists in selecting a fixed number of vertices from each graph and exploiting a canonical ordering on graph vertices. For each vertex, it defines a fixed-size neighborhood, exploiting the same vertex ordering. It requires

the vertices of each input graph to be in a canonical ordering, which is as complex as the graph isomorphism problem (no polynomial-time algorithm is known).

The Funnel GCNN (FGCNN) model [33] aims to enhance the gradient propagation using a simple aggregation function and LeakyReLU activation functions. Hinging on the similarity of the adopted graph convolutional operator, that is, the *GraphConv* (see Section 14.5.3), to the Weisfeiler–Lehman (WL) Subtree Kernel [34], it introduces a loss term for the output of each convolutional layer to guide the network to reconstruct the corresponding explicit WL features. Moreover, the number of filters used at each convolutional layer is based on a measure of the WL kernel complexity.

Bianchi et al. [35] developed a convolutional layer based on auto-regressive moving average (ARMA) filters. This particular type of filter, compared to polynomial filters, allows to manage higher order neighborhoods. Moreover, ARMA filters present a flexible frequency response. The resulting nonlinear graph filter enhanced the modeling capability compared to common GNNs that exploit convolutional layers based on polynomial filters.

Recently, geometric graph convolutional networks (Geom-GCN) [36] were proposed. The model exploits a geometric aggregation scheme that is permutation-invariant and consists of three modules: node embedding, structural neighborhood, and bi-level aggregation. For what concerns the node embedding component, the authors proposed to use three different ad-hoc embedding methods to compute node embeddings that preserve specific topology patterns. The structural neighborhood consists in the definition of a neighborhood that is the result of the concatenation of the neighborhood in the graph space and the neighborhood in the latent space. Specifically, the neighborhood of a node v in the latent space is the set of nodes whose distance from v is less than a predefined value. To compute the distance, a particular function is defined over two latent space node representations, and it is based on a similarity measure between nodes in the latent space. The bi-level aggregation exploits two aggregation functions used in defining the graph neural network layer. These two functions guarantee permutation invariance for a graph, and they are able to extract effectively structural information of nodes in neighborhoods.

14.5.10. *Beyond the Message Passing*

Some recent works in the literature exploit the idea of extending graph convolution layers to increase the receptive field size, without increasing the depth of the model. The basic idea underpinning these methods is to consider the case in which the graph convolution can be expressed as a polynomial of the powers of a transformation $\mathcal{T}(\cdot)$ of the adjacency matrix. The models based on this idea are able to simultaneously and directly consider all topological receptive fields up to k-hops, just like the ones that

are obtained by a stack of graph convolutional layers of depth k, without incurring the typical limitations related to the complex interactions among the parameters of the GC layers. Formally, the idea is to define a representation as built from the contribution of all topological receptive fields up to k-hops as

$$\mathbf{H} = f(\mathcal{T}(\mathbf{A})^0\mathbf{X}, \mathcal{T}(\mathbf{A})^1\mathbf{X}, \ldots, \mathcal{T}(\mathbf{A})^k\mathbf{X}), \tag{14.29}$$

where $\mathcal{T}(\cdot)$ is a transformation of the adjacency matrix (e.g., the Laplacian matrix), and f is a function that aggregates and transforms the various components obtained from the powers of $\mathcal{T}(\mathbf{A})$, for instance the concatenation, the summation, or even something more complex, such as a multilayer perceptron. The f function can be defined as a parametric function, depending on a set of parameters θ whose values can be estimated from data (e.g., when f involves an MLP). In the following, we discuss some recent works that define instances of this general architectural component.

One of the first methods that exploited this intuition is presented by Atwood and Towsley [37]. In their work, the authors propose two architectures designed to perform node classification and graph classification (we will discuss in depth the distinction between the two tasks in Section 14.7). Specifically, Atwood and Towsley proposed to exploit the power series of the probability transition matrix, multiplied (using the Hadamard product) by the inputs. Moreover, the model developed to perform graph classification exploits the summation as f function. Similarly, the model proposed by Defferrard et al. in 2016 [18] exploits the Chebyshev polynomials and sums the obtained representations over k.

Another interesting approach is that proposed by Tran et al. [38], where the authors consider larger receptive fields compared to standard graph convolutions. They focus on a convolution definition based on the shortest paths, instead of the standard random walks obtained by exponentiation of the adjacency matrix.

Wu et al. introduced a simplification of the graph convolution operator, dubbed Simple Graph Convolution (SGC) [39]. The proposed model is based on the idea that perhaps the nonlinear operator introduced by GCNs is not essential. The authors propose to stack several linear GC operators, since stacking multiple GC layers has an important effect on the location of the learned filters (after k GC layers, the hidden representation of a vertex considers information coming from the vertices up to distance k). Formally, the model is defined as follows: $\mathbf{Y} = softmax(\mathbf{S}^k\mathbf{X}\Theta)$, where $\mathbf{S} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$, $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, and $\tilde{\mathbf{D}}$ is the degree matrix of $\tilde{\mathbf{A}}$. Notice that, in this case, f selects just the kth power of the diffusion operators. An extension of this work, dubbed Linear Graph Convolution (LGC), was proposed by Navarin et al. [21], where the authors propose to introduce skip connections from each layer to the last one, which is a merge layer implementing the sum operator, followed by a softmax activation $\mathbf{Y} = softmax(\sum_{i=0}^{k} \alpha_i \mathbf{L}^i\mathbf{X}\Theta)$.

Liao et al. in 2019 [40] proposed to construct a deep graph convolutional network, exploiting particular localized polynomial filters based on the Lanczos algorithm, which leverages multiscale information.

In the same year, Chen et al. [41] suggested to replace the neighbor aggregation function by graph-augmented features combining node degree features and multiscale graph-propagated features. Basically, the proposed model concatenates the node degree with the power series of the normalized adjacency matrix. The proposed model aggregates the graph-augmented features of each vertex and projects each of these subsets by using an MLP.

Luan et al. [42] introduced two deep GCNs that rely on Krylov blocks. The first one exploits a GC layer, named snowball, which concatenates multiscale features incrementally, resulting in a densely connected graph network. The architecture stacks several layers, and exploits nonlinear activation functions. The second model, called Truncated Krylov, concatenates multiscale features in each layer. In this model, the topological features from all levels are mixed together. A similar approach is proposed by Rossi et al. [43]. The authors proposed an alternative method, named SIGN, to scale GNNs to very large graphs. This method uses, as a building block, the set of exponentiations of linear diffusion operators. In this building block, every exponentiation of the diffusion operator is linearly projected by a learnable matrix. Moreover, a nonlinear function is applied to the concatenation of the diffusion operators.

Very recently, a model dubbed deep adaptive graph neural network was introduced [44], to learn node representations by adaptively incorporating information from large receptive fields. The model exploits an MLP network for node feature transformation. Then, it constructs a multiscale representation leveraging on the computed node feature transformations and the exponentiation of the adjacency matrix. This representation is obtained by stacking the various adjacency matrix exponentiations (thus obtaining a three-dimensional tensor). Similarly to the GCNs that rely on Krylov blocks [42], also in this case the model projects the obtained multiscale representation using multiplication by a weights matrix, obtaining a representation where the topological features from all levels are mixed together. Moreover, this projection also uses (trainable) retention scores. These scores measure how much information on the corresponding representations derived by different propagation layers should be retained to generate the final representation for each node in order to adaptively balance the information from local and global neighborhoods.

14.5.11. *Relational Data*

In some applications, edges are typed, i.e., edges come with associated labels. This is the case, for instance, with knowledge graphs, where the edges specify the

relationship between two concepts. This kind of data is, thus, commonly addressed as relational data.

Graph neural networks can be easily extended to deal with such data [45]. Relational Graph Convolutional Networks are defined for a node v as

$$\mathbf{h}_v^{(i+1)} = \sigma\left(\mathbf{W}^{(i)}\mathbf{h}_v^{(i)} + \sum_{r\in\mathcal{R}}\sum_{u\in\mathcal{N}_r(v)}\frac{1}{|\mathcal{N}_r(v)|}\mathbf{W}_r'^{(i)}\mathbf{h}_u^{(i)}\right), \tag{14.30}$$

where $\mathcal{R}$ is the set of possible relations, $\mathcal{N}_r(v)$ is the set of neighbors of v that are connected to it via an edge of type r, and $\mathbf{h}_v^{(0)} = \mathbf{x}_v$. When the set of possible relations is big, authors suggest to use regularization to constrain $\mathbf{W}_r'^{(i)}$.

Simonovsky and Komodakis [46] proposed a similar approach which was an extension of the model proposed by Duvenaud et al. [47] (see Section 14.3.2) that computes the sum over the neighbors of a vertex by weights conditioned by the edge labels.

14.6. Expressive Power of Graph Neural Networks

When dealing with graphs, the problem of determining whether two graphs are just different representations of the same one is known as the graph isomorphism problem [48], for which there are no known polynomial-time algorithms. This means that, for any graph neural network, there exist infinitely many non-isomorphic graphs that the model cannot distinguish, i.e., that will be mapped to the same representation. This is a central issue for graph neural networks, since the more indistinguishable graphs there are, the more limited is the class of functions the network can represent (i.e., the network can only represent functions that associate the same output to all the graphs that are mapped to the same internal representation).

The expressiveness of graph neural networks has been recently studied [49]. The authors did show that most graph neural networks are at most as powerful as the one-dimensional *Weisfeiler–Lehman* graph invariant (see Section 14.5.3), depending on the adopted neighbor aggregation function. They proved that some GNN variants, such as the GCN described in Section 14.5.2, or graphSAGE described in Section 14.5.7, do not achieve this expressiveness, and proposed the graph isomorphism network (GIN) model, which was proven to be as expressive as the *Weisfeiler–Lehman* test. In GIN, the aggregation over node neighbors is implemented using an MLP; therefore, the resulting GC formulation is the following (where $\mathbf{h}_v^{(0)} = \mathbf{x}_v$):

$$\mathbf{h}_v^{(k)} = MLP^{(k)}((1+\epsilon^{(k)})\mathbf{h}_v^{(k-1)} + \sum_{u\in\mathcal{N}(v)}\mathbf{h}_u^{(k-1)}). \tag{14.31}$$

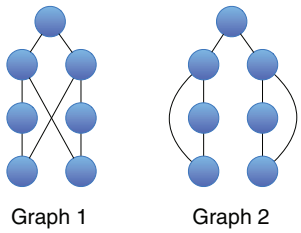


Figure 14.2: Example of two non-isomorphic graphs that cannot be distinguished by the 1-dimensional *Weisfeiler–Lehman* graph invariant test. Note that, in this simple example, all nodes have the same label.

14.6.1. *More Expressive Graph Neural Networks*

While the 1-dimensional *Weisfeiler–Lehman* graph invariant test is a fast and generally effective graph isomorphism test, there are many (even small) pairs of graphs it cannot distinguish. Figure 14.2 reports a pair of graphs that are not isomorphic but that cannot be distinguished by 1-WL test.

One possibility to define more expressive graph neural networks is to inspire their computation to more powerful graph invariants, such as the k-dimensional *Weisfeiler–Lehman* (k-WL) tests, which base the iterative coloring procedure of WL on graphlets of size k. k-WL tests are increasingly powerful for all $k > 2$ ($k = 1$ and $k = 2$ have the same discriminative power).

Some recent works followed this direction. Maron et al. [50] proposed a hierarchy of increasingly expressive graph neural networks, where the k-th order network is as expressive as the k-dim WL. A practical implementation of a network as expressive as the three-dimensional WL is then proposed. The main drawback of the proposed approach is the computational complexity that is quadratic in the number of nodes of the input graphs (even for graphs with sparse connectivity). Moreover, such expressive networks tend to easily overfit the training data.

Morris et al. [51] proposed a more efficient alternative, defining a revisited, local version of the k-WL that exploits the graph sparsity. They proposed a neural network architecture based on such a method and showed that it improves the generalization capability of the network compared to other alternatives in the literature. Again, the main problem of the approach is that to compute the k-order node tuples on which the WL test is based, the computational complexity is $O(n^k)$.

14.7. Prediction Tasks for Graphs

When considering graph-level prediction tasks, the node-level representations computed by the graph convolutional operators need to be aggregated in order to obtain a single (fixed-size) representation of the graph. Thus, one of the main

problems to solve in this scenario is how to transform a variable number of node-level representations into a single graph-level one. Formally, a general GNN model for graph classification (or regression) is built according to the equations described in the following. First, d graph convolution layers are stacked

$$\mathbf{H}^{(i)} = \sigma(GC(\mathbf{H}^{(i-1)}, G)), \tag{14.32}$$

where $\sigma(\cdot)$ is an element-wise nonlinear activation function, $GC(\cdot,\cdot)$ is a graph convolution operator, and $\mathbf{h}_v^{(i)}$ (the v-th row of $\mathbf{H}^{(i)}$) is the representation of node v at the i-th graph convolution layer, $1 \leq i \leq d$, and $\mathbf{h}_v^{(0)} = \mathbf{x}_v$. Then, an aggregation function is applied

$$\mathbf{h}^S = aggr(\{\mathbf{h}_v^{(i)} |\ v \in V,\ 1 \leq i \leq d\}), \tag{14.33}$$

where $aggr(\cdot)$ is the aggregator function. Note that the aggregation may depend on all the hidden representations computed by the different GC layers and not just the last one. $\mathbf{h}^S$ is the fixed-size graph-level representation. Subsequently, the $readout(\cdot)$ function (usually implemented by a multilayer perceptron) applies some nonlinear transformation on $\mathbf{h}^S$. Finally, an output layer (e.g., the *LogSoftMax* for a classification problem) is applied.

14.7.1. *Aggregation Function and Readout*

As we highlighted above, the GC can be used to perform either graph classification or node classification. Therefore, the main additional component required by graph classification GNNs is the aggregator function. An effective and efficient graph-level representation should be, as much as possible, invariant to different isomorphic representations of the input graph, thus letting the learning procedure to focus only on the property prediction task, with no need to worry about the way the graph in the input is represented. The simplest aggregation operators adopted in the literature are linear, namely the average and the sum of vertex representations. This kind of operators have been used in many GNN models. For instance, NN4G [15] (described in Section 14.3.2) computes, for each graph, the average graph vertex representation for each hidden layer, and concatenates them. Other approaches consider only the last graph convolution layer to compute such an average [37]. A more complex approach exploits multilayer perceptrons to transform node representations before a sum aggregator is applied [19].

In the last few years, several more complex (nonlinear) techniques to perform aggregation have been proposed. One of these is *SortPooling* [31], which is a nonlinear pooling operator, used in conjunction with concatenation to obtain an aggregation operator. The idea is to select a predetermined number of vertex

embeddings using a sorting function, and to concatenate them, obtaining a graph-level representation of fixed size. Note, however, that this representation ignores some of the nodes of the graph, thus losing potentially important information.

In many cases, using such simple aggregators inevitably results in a loss of information due to the mix of numerical values they introduce. A much better approach, from a conceptual point of view, would be to consider all the representations of nodes of a graph as a (multi)set, and the aggregation function to learn as a function defined on these (multi)sets. To face this setting, the DeepSets [52] network has been recently proposed. The idea is to design neural networks that take sets as the input. A DeepSet maps the elements of the input set in a high-dimensional space via a learned $\phi(\cdot)$ function, usually implemented as a multilayer perceptron. It then aggregates node representations by summing them, and finally it applies the readout, i.e., the $\rho(\cdot)$ function (another MLP), to map the graph-level representation to the output of the task at hand. Formally, a DeepSets network can be defined as follows:

$$sf(X) = \rho\left(\sum_{x_i \in X} \phi(x_i)\right), \tag{14.34}$$

for some $\rho(\cdot)$ and $\phi(\cdot)$ functions, if X is countable. Navarin et al. [53] proposed a graph aggregation scheme based on DeepSets implementing the $\phi(\cdot)$ function as a multilayer perceptron. An interesting property of this approach is that it has been proven that any function $sf(X)$ over a set X satisfying the following two properties:

(1) variable number of elements in the input, i.e., each input is a set $X = \{x_1, \ldots, x_m\}$ with x_i belonging to some set $\mathcal{X}$ (typically a vectorial space) and $M > 0$;
(2) permutation invariance;

is decomposable in the form of Eq. (14.34).

Moreover, under some assumptions, DeepSets are universal approximators of functions over countable sets, or uncountable sets with a fixed size. Therefore, they are potentially very expressive from a functional point of view.

Recently, Pasa et al. [54] proposed to extend this approach by implementing $\phi(\cdot)$ by exploiting self-organizing maps (SOMs) to map the node representations in the space defined by the activations of the SOM neurons. The resulting representation embeds the information about the similarity between the various inputs. In fact, similar input structures will be mapped in similar output representations (i.e., node embeddings). Using a fully unsupervised mapping for the $\phi(\cdot)$ function may lead to the loss of task-related information. To avoid this issue, the proposed method makes the $\phi(\cdot)$ mapping supervised by stacking, after the SOM, a graph convolution layer

that can be trained via supervised learning, allowing to better incorporate topological information in the mapping.

14.7.2. *Graph Pooling*

Pooling operators have been defined and used in convolutional neural networks (CNNs) for image processing. Their aim is to improve the performance of the network by introducing some degree of local invariance into transformations of the image, such as scale, translation, or rotation transformations. Replicating these operators in the context of graphs turns out to be very complex for different reasons. First, in the GNN setting—compared to standard CNNs—graphs contain no natural notion of spatial locality because of the complex topological structure. Moreover, unlike image data, in the graph context the number and degree of nodes are not fixed, thus making it even more complex to define a general graph pooling operator. In order to address these issues, different types of graph pooling operators have been proposed in the last two years. In general, graph pooling operators are defined so as to learn a clustering of nodes that allows to uncover the hierarchical topology given by the underlying sub-graphs. In a GNN architecture, the graph pooling operators can be inserted after one or more GC layers.

One example of the first proposed graph pooling operators is DiffPool [55]. DiffPool computes a node clustering by learning a cluster assignment matrix $S^{(l)}$ over the node embedding computed after l GC layers. Let us denote the adjacency matrix at the l-th layer as $A^{(l)}$ and the node embedding matrix as $Z^{(l)}$. DiffPool computes $A^{(l+1)}$, $X^{(l+1)}$, which are, respectively, the new adjacency matrix and the new matrix of embeddings for each of the nodes/clusters in the graph. $X^{(l+1)}$ is computed by aggregating the nodes embeddings $Z^{(l)}$ according to the cluster assignments $S^{(l)}$, generating embeddings for each of the clusters:

$$X^{(l+1)} = {S^{(l)}}^{\top} Z^{(l)}.$$

Similarly, the new adjacency matrix $A^{(l+1)}$ represents the connections between clusters, and is computed as

$$A^{(l+1)} = {S^{(l)}}^{\top} A^{(l)} S^{(l)}.$$

The main limitation of DiffPool is the computation of the soft clustering assignments, since during the early phases of training, a dense assignment matrix must be stored in memory. Asymptotically, this incurs a quadratic storage complexity over the number of graph vertices, making the application of DiffPool in large graphs unfeasible.

To face this complexity issue, a sparse version of graph pooling has been proposed [56], where a differentiable graph coarsening method is used to reduce the size of the graph in an adaptive manner within a graph neural network pipeline.

The sparsity of this method is achieved by applying a node dropping policy, that basically drops a fixed number of nodes from the original graph. The nodes to drop are selected based on a *projection score* computed for each node. This method allows to reduce the memory storage complexity, which becomes linear in the number of edges and nodes of the graph.

Another recently proposed graph pooling method is MinCutPool [57]. The method is based on the *k-way normalized minCUT* problem, which that is the task of partitioning V in k disjoint subsets by removing the minimum number of edges. The idea is to learn the parameters in a MinCutPool layer by minimizing the minCUT objective, which can be jointly optimized with a task-specific loss. The method, similarly to DiffPool, uses a cluster assignment matrix. This matrix is computed by solving a relaxed continuous formulation of the minCUT optimization problem that can be solved in polynomial time and guarantees a near-optimal solution.

14.8. Experimental Comparison

In this section, we revise some of the most interesting results obtained by GNN models in different tasks. First, we outline the main applications of GNNs and the dataset commonly used as benchmarks to evaluate the strengths and the weaknesses of the models. Then, we discuss a selection of the results obtained by models that achieved notable results.

14.8.1. *Applications and Benchmark Datasets*

Graph structured data are ubiquitous in nature; therefore, there are a wide number of possible real-world applications of graph neural networks. However, it is important to take into account that in the past some problems were usually modeled using flat or sequential representations, instead of using graphs. This is due to the fact that for some tasks involving graph structured data, considering the topological information is not crucial, and using a simpler model may bring benefits in terms of performance. In this section, we list and discuss the most interesting tasks on which GNNs have been successfully applied in the last few years.

One of the most prominent areas in which GNNs have been applied with success is Cheminformatics. In fact, many chemical compound datasets are often used to benchmark new models. In Cheminformatics, different tasks can be considered. In general, the input is the molecular structure, while the output varies based on the specific task. The majority of tasks in benchmark datasets model graph classification or regression problems. An example is the quantitative structure–property relationship (QSPR) analysis, where the aim is to predict one or more chemical properties (e.g., toxicity, solubility) of the input chemical compound. Other tasks in this field include finding structural similarities among compounds, drug side-effect identification,

and drug discovery. The datasets related to these applications, commonly used as benchmarks, are MUTAG [58], PTC [59], NCI1 [60], PROTEINS, [61], D&D [62], and ENZYMES [61]. The first two datasets contain chemical compounds represented by their molecular graphs, where each node is labeled with an atom type, and the edges represent bonds between them. MUTAG contains aromatic and hetero-aromatic nitro compounds, and the task is to predict their mutagenic effect on a bacterium. PTC contains chemical compounds and the task is to predict their carcinogenicity for male rats. In NCI1, the graphs represent anti-cancer screens for cell lung cancer. The last three datasets, PROTEINS, D&D, and ENZYMES, contain graphs that represent proteins. Each node corresponds to an amino acid, and an edge connects two of them if they are less than 6Å apart. In particular, ENZYMES, unlike the other reported datasets (that model binary classification problems), allows testing the models on multiclass classification over six classes.

Another interesting field where GNNs have achieved good results is social network analysis. A social network is modeled as a graph where nodes represent users, and edges the relationships between them (e.g., friendship, co-authorship). Usually, the tasks on social graphs regard node or graph classification. For what concerns node classification, three major datasets in the literature are usually employed to assess the performances of GNNs: Cora [63], Citeseer [64], and PubMed [65]. Each dataset is represented as a single graph. Nodes represent documents, and node features are sparse bag-of-words feature vectors. Specifically, the task requires us to classify the research topics of papers. Each node represents a scientific publication described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from a dictionary. For what concerns graph classification on social datasets, the most popular social benchmarks are COLLAB, IMDB-BINARY (IMDB-B), IMDB-MULTI (IMDB-M), REDDIT-BINARY, and REDDIT-MULTI [66]. In COLLAB, each graph represents a collaboration network of the corresponding researcher with other researchers from three fields of physics. The task consists in predicting the physics field the researcher belongs to. IMDB-B and IMDB-M are composed of graphs derived from actors/actresses who have acted in different movies on IMDB, together with the movie genre information. Each graph has a target that represents the movie genre. IMDB-B models a binary classification task, while IMDB-M contains graphs that belong to three different classes. Unlike the bioinformatics datasets, the nodes contained in the social datasets do not have any associated label. REDDIT-BINARY and REDDIT-MULTI contain graphs that represent online discussion threads where nodes represent users, and an edge represents the fact that one of the two users responded to the comment of the other user. In REDDIT-BINARY, four popular subreddits are considered. Two of them are question/answer-based subreddits, while the other two are discussion-based subreddits. A graph is labeled according to whether it belongs to a question/answer-based community

or a discussion-based community. By contrast, in REDDIT-MULTI, there are five subreddits involved and the graphs are labeled with their corresponding subreddits.

14.8.2. *Experimental Setup and Validation Process*

In the last few years, many new graph neural network models have been proposed. However, despite the theoretical advancements reached by the latest contributions in the field, comparing the results of the various proposed methods turns out to be hard. Indeed, many different methods (not all correct) to perform validation, and thus select the model's hyper-parameters, were used. In a recently published paper [67], the authors highlighted and discussed the importance of the validation strategy, especially when dealing with graph neural networks. Moreover, they experimentally proved that the experimental settings of many papers are ambiguous or not reproducible. Another important issue is related to the correct usage of data splits for model selection versus model assessment. A fair method to evaluate a model requires two distinct phases: model selection on the validation set, and model assessment on the test set. Unfortunately, in many works, in particular, in graph classification, some of the hyper-parameter are selected by considering the best results directly on the test set, clearly an incorrect procedure. This is also due to the fact that some datasets have a limited number of samples, in particular in validation, and thus a hyper-parameter selection performed using the validation set is extremely unstable. However, performing model selection and model assessment in an incorrect way could lead to overly optimistic and biased estimates of the true predictive performance of a model. Another aspect that should be considered comparing two or more models is related to the amount of information used in input and output encoding. In fact, it is common practice in the literature to augment node descriptors with structural features. For example, some models add the degree (and some clustering coefficients) to each node feature vector [55], or add a one-hot representation of node degrees [49]. Obviously, this encoding difference makes it even harder to compare the experimental results published in the literature. Good experimental practices suggest that all models should be consistently compared using the same input representations, and using the same validation strategy. For this reason, the results reported in the following section are limited to models that have been evaluated using the same, fair methodology.

14.8.3. *Experimental Results*

In this section, we report and discuss the most meaningful results obtained by some of the models discussed in the previous sections. Table 14.1 reports the results on classification tasks over bioinformatics benchmark datasets, while in Table 14.2 we report the results on classification tasks over social network datasets.

Table 14.1: Accuracy comparison among several state-of-the-art models on graph classification tasks

Model\Dataset	**PTC**	**NCI1**	**PROTEINS**	**D&D**	**ENZYMES**
PSCN [32]	60.00 ±4.82	76.34 ±1.68	75.00 ±2.51	76.27 ±2.64	—
FGCNN [53]	58.82 ±1.80	81.50 ±0.39	74.57 ±0.80	77.47 ±0.86	— —
SOM-GCNN [54]	62.24 ±1.7	83.30 ±0.45	75.22 ±0.61	78.10 ±0.60	50.01 ±2.9
DGCNN [67]	— —	76.4 ±1.7	72.9 ±3.5	76.6 ±4.3	38.9 ±5.7
GIN [67]	— —	80.0 ±1.4	73.3 ±4.0	75.3 ±2.9	59.6 ±4.5
DIFFPOOL [67]	— —	76.9 ±1.9	73.7 ±3.5	75.0 ±3.5	59.5 ±5.6
GraphSAGE [67]	— —	76.0 ±1.8	73.0 ±4.5	72.9 ±2.0	58.2 ±6.0

Table 14.2: Accuracy comparison among several state-of-the-art models on graph classification tasks considering social network datasets

Model\Dataset	**COLLAB**	**IMDB-B**	**IMDB-M**	**RED.-B**	**RED.-5K**
PSCN [32]	72.60 ±2.15	71.00 ±2.29	45.23 ±2.84	86.30 ±1.58	49.10 ±0.70
DGCNN [67]	57.4 ±1.9	53.3 ±5.0	38.6 ±2.2	72.1 ±7.8	35.1 ±1.4
GIN [67]	75.9 ±**1.9**	66.8 ±3.9	42.2 ±4.6	87.0 ±4.4	53.8 ±5.9
DIFFPOOL [67]	67.7 ±1.9	68.3 ±6.1	45.1 ±3.2	76.6 ±2.4	34.6 ±2.0
GraphSAGE [67]	71.6 ±1.5	69.9 ±4.6	47.2 ±3.6	86.1 ±2.0	49.9 ±1.7

Finally, Table 14.3 reports the most interesting results in the literature for what concerns the task of node classification. The results reported in the tables show that there isn't a method that obtains the best performance on all the considered datasets/tasks. This behavior suggests that each of the considered datasets and

Table 14.3: Accuracy comparison among several state-of-the-art models on nodes classification tasks considering social network datasets

Model\Dataset	**Citeseer**	**Cora**	**PubMed**
Cheby [68]	70.2 ±1.0	81.4 ±0.7	78.4 ±0.4
GCN [68]	71.1 ±0.7	81.5 ±0.6	79.0 ±0.6
GAT [68]	70.8 ±0.5	83.1 ±0.4	78.5 ±0.3
SGC [68]	71.3 ±0.2	81.7 ±0.1	78.9 ±0.1
ARMA [68]	72.3 ±1.1	82.8 ±0.6	78.8 ±0.3
LGC [21]	72.9 ±0.3	85.0 ±0.3	80.2 ±0.4
APPNP [68]	71.8 ±0.5	83.3 ±0.5	80.1 ±0.2
GCNII [26]	73.4	85.5	80.3

tasks stress different aspects of the graph convolution operators. For what concerns graph classification and bioinformatics datasets, the best results are obtained by the SOM-GCNN in all datasets except ENZYMES, where GIN, obtains the best result. A very similar accuracy is achieved by DIFFPOOL. These results emphasize the importance of using an expressive aggregation and readout methodology. Indeed SOM-GCNN, GIN, and DIFFPOOL introduce novel expressive aggregation methodologies. The social network results in Table 14.2 show that even in these datasets, GIN obtains very interesting performances since it obtains the best results in three datasets: COLLAB, and two versions of REDDIT (binary classification and multiclass). These datasets have a significantly higher number of edges and nodes per graph than the two IMDB datasets, where PSCN obtains a better accuracy. In node classification tasks results (Table 14.3), the GCNII shows the best performances in all datasets. In Cora, the accuracy obtained by GCNII is very close to the one obtained by LGC. In Citeseer, GCNII performs 1% better than LGC, which achieved the second highest result. In PubMed, the accuracy values of GCNII, LGC, and APPNP are very close; indeed the performances of the three methods are less than one standard deviation apart.

14.9. Open-Source Libraries

The growth in popularity of GNN models poses new challenges for the existing computing libraries and frameworks aimed at developing DNN models. In fact, the particular representations of the nodes and of the diffusion operators make it necessary to handle the sparsity of GNN operations efficiently in widespread GPU hardware. Moreover, considering the enormous size of some graphs (e.g., social networks datasets) it became crucial to implement the possibility of scaling the computations to large-scale graphs and multiple GPUs. For these reasons, some new libraries that extend and adapt the functionality of widely adopted DNN frameworks, such as PyTorch [69] and TensorFlow [70], have been recently developed. These libraries provide high-level interfaces and methods to implement and develop multiple GNN variants. In the following section, we review three commonly adopted libraries that are built on the top of PyTorch and TensorFlow to develop GNN models: PyTorch Geometrics, Deep Graph Library, and Spektral.

14.9.1. *Pytorch-Geometric*

PyTorch Geometric (PyG) [68] is a widespread library that is built upon PyTorch. PyG includes various methods for deep learning on graphs and other irregular structures, from a variety of published papers. The library allows to easily develop GNN architectures thanks to an interface that implements message passing. This interface presents methods to define the message, update the node embeddings, and perform neighborhood aggregation and combination. Moreover, multiple pooling operations are provided. For what concerns the data management, the library implements a mini-batch loader working either for many small graphs, or when the input is a single giant graph. Finally, the library implements loaders and management functions for the most common benchmark datasets.

PyG exploits dedicated GPU scatter and gather kernels to handle sparsity, instead of using sparse matrix multiplication kernels. The scatters allow to operate on all edges and nodes in parallel, thus significantly accelerating the GNN processing.

14.9.2. *Deep Graph Library*

Deep Graph Library (DGL) [71] is a framework-agnostic library. In fact, it can use PyTorch, TensorFlow, or MXNet [72] as backends. Similarly to PyG, DGL provides a message passing interface. The interface exposes three main methods: the message function to define the message; the reduce function that allows each node to access the received messages and perform aggregation; and the update function that operates on the aggregation results. This function is typically used to combine the aggregation with the original features of a node and to compute the updated node embedding.

To improve the performance in graph processing, DGL adopts advanced optimization techniques such as kernel fusion, multithread and multiprocess acceleration, and automatic sparse format tuning. In particular, it leverages on specialized kernels for GPUs or TPUs that allow to improve the parallelization of matrix multiplications between both dense and sparse matrices. The adopted parallelization scheme is chosen by the library using heuristics that consider multiple factors including the input graph. Thanks to these optimizations, compared to other popular GNN frameworks, DGL shows better performance both in terms of computational time demand and memory consumption.

14.9.3. *Spektral*

Very recently, a new library dubbed Spektral [73] has been introduced. The Spektral library is based on the Keras API and TensorFlow 2. The library provides the essential building blocks for creating graph neural networks. Moreover, it implements some of the most common GC layers as Keras layers. This allows to use them to build Keras models, and thus they inherit the most important features of Keras such as the training loop, callbacks, distributed training, and automatic support for GPUs and TPUs. At the time of writing, Spektral implements 15 different GCN layers, based on the message passing paradigm. Furthermore, the library provides several graph pooling layers and global graph pooling methods. Comparing Spektral with PyG and DGL, it is important to notice that Spektral is developed specifically for the TensorFlow ecosystem. In terms of computational performance, Spektral's implementation of GC layers is comparable to the one of PyG.

14.10. Conclusions and Open Problems

In this chapter, we introduced and discussed graph neural networks (GNNs). We firstly introduced the main building block of GNNs: the graph convolution (GC). We discussed how this component is exploited by graph neural networks to compute a (hopefully) sound and meaningful representation for graph nodes, and how GC operators evolved from the first definitions to the GC layers recently proposed in the literature. The experimental results obtained using GNNs show the benefits of using this type of model when dealing with structured data. Despite the good results obtained in several graph-based applications, there are several open problems and challenges. One of the main open problems is scalability. Several models that obtain very good results in benchmark datasets turn out to be not scalable to large-scale settings. Developing models capable of dealing with very large graphs is crucial, since several interesting tasks require dealing with huge amounts of data (e.g., social networks).

In addition, from a theoretical prospective, graph neural networks offer very interesting and challenging open problems. Indeed, the expressiveness of many graph neural networks is rather limited. As we discussed, recent works showed that GNNs are as powerful as the one-dimensional Weisfeiler–Lehman graph invariant test. Only recently, a few models that try to overcome this limit have been introduced. The encouraging results highlight that more research has to be carried out in this direction.

From the architectural point of view, GNNs belong to the deep learning framework, but most of the models in the literature use a very number amount of stacked convolutional layers. Going deep into the number of layers leads to some known issues such as the complex gradient propagation through the layers, and the over-smoothing problem [25]. Exploiting deeper models would allow obtaining richer representations of graph nodes to be obtained; therefore, finding a model that allows to develop deeper GNNs, could help to further improve the state-of-the-art performances.

References

[1] A. Sperduti, D. Majidi, and A. Starita, Extended Cascade-Correlation for syntactic and structural pattern recognition. In eds. P. Perner, P. S. Wang, and A. Rosenfeld, *Advances in Structural and Syntactical Pattern Recognition, 6th International Workshop, SSPR '96,* Leipzig, Germany, August 20–23, 1996*, Proceedings*, vol. 1121, *Lecture Notes in Computer Science*, pp. 90–99. Springer (1996). doi: 10.1007/3-540-61577-6_10. URL https://doi.org/10.1007/3-540-61577-6_10.

[2] A. Sperduti and A. Starita, Supervised neural networks for the classification of structures, *IEEE Trans. Neural Networks*, **8**(3), 714–735 (1997).

[3] M. M. Bronstein, J. Bruna, Y. Lecun, A. Szlam, and P. Vandergheynst, Geometric deep learning: going beyond Euclidean data, *IEEE Signal Process. Mag.*, **34**(4), 18–42 (2017). ISSN 10535888.

[4] P. Frasconi, M. Gori, and A. Sperduti, A general framework for adaptive processing of data structures, *IEEE Trans. Neural Networks*, **9**(5), 768–786 (1998). doi: 10.1109/ 72.712151. URL https://doi.org/10.1109/72.712151.

[5] M. Collins and N. Duffy, Convolution kernels for natural language. In *Proceedings of the Advances in Neural Information Processing Systems,* vol. 14, pp. 625–632 (2001).

[6] R. I. Kondor and J. Lafferty, Diffusion kernels on graphs and other discrete structures. In *ICML* (2002).

[7] N. Shervashidze, P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt, Weisfeiler-Lehman graph kernels, *JMLR*, **12**, 2539–2561 (2011).

[8] G. Da San Martino, N. Navarin, and A. Sperduti, Ordered decompositional DAG kernels enhancements, *Neurocomputing*, **192**, 92–103 (2016).

[9] A. Feragen, N. Kasenburg, J. Petersen, M. de Bruijne, and K. M. Borgwardt, Scalable kernels for graphs with continuous attributes. In *Neural Information Processing Systems (NIPS) 2013*, pp. 216–224 (2013).

[10] G. Da San Martino, N. Navarin, and A. Sperduti, Tree-based kernel for graphs with continuous attributes, *IEEE Trans. Neural Networks Learn. Syst.*, **29**(7), 3270–3276 (2018). ISSN 2162-237X.

[11] G. Nikolentzos, G. Siglidis, and M. Vazirgiannis, Graph kernels: a survey, *CoRR*, **abs/1904.12218** (2019). URL http://arxiv.org/abs/1904.12218.

[12] N. M. Kriege, F. D. Johansson, and C. Morris, A survey on graph kernels, *Appl. Network Sci.,* **5**(1), 6 (2020). doi: 10.1007/s41109-019-0195-3. URL https://doi.org/10.1007/s41109-019-0195-3.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks. In eds. P. L. Bartlett, F. C. N. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3–6 (2012)*, Lake Tahoe, Nevada, United States, pp. 1106–1114 (2012). URL https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[14] F. Scarselli, M. Gori, A. C. Ah Chung Tsoi, M. Hagenbuchner, and G. Monfardini, The graph neural network model, *IEEE Trans. Neural Networks*, **20**(1), 61–80 (2009).

[15] A. Micheli, Neural network for graphs: a contextual constructive approach, *IEEE Trans. Neural Networks*, **20**(3), 498–511 (2009).

[16] L. B. Almeida, A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In eds. M. Caudil and C. Butler, *Proceedings of the IEEE First International Conference on Neural Networks*, San Diego, CA, pp. 609–618 (1987).

[17] F. J. Pineda, Generalization of back-propagation to recurrent neural networks, *Phys. Rev. Lett*., **59**(19), 2229–2232 (1987). ISSN 1079-7114. doi: 10.1103/PhysRevLett.59.2229. URL https://link.aps.org/doi/10.1103/PhysRevLett.59.2229.

[18] M. Defferrard, X. Bresson, and P. Vandergheynst, Convolutional neural networks on graphs with fast localized spectral filtering. In *Neural Information Processing Systems (NIPS)*, pp. 3844–3852 (2016).

[19] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning*, pp. 1263–1272 (2017).

[20] D. I. Shuman, S. K. Narang, P. Frossard, A. Ortega, and P. Vandergheynst, The emerging field of signal processing on graphs: extending high-dimensional data analysis to networks and other irregular domains, *IEEE Signal Process. Mag.,* **30**(3), 83–98 (2013).

[21] N. Navarin, W. Erb, L. Pasa, and A. Sperduti, Linear graph convolutional networks. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, pp. 151–156 (2020).

[22] T. N. Kipf and M. Welling, Semi-supervised classification with graph convolutional networks. In *ICLR*, pp. 1–14 (2017).

[23] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, Weisfeiler and Leman go neural: higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 4602–4609 (2019).

[24] J. Klicpera, A. Bojchevski, and S. Günnemann, Predict then propagate: graph neural networks meet personalized pagerank. In *7th International Conference on Learning Representations, ICLR 2019,* New Orleans, LA, USA, May 6–9 (2019). OpenReview.net (2019).

[25] K. Oono and T. Suzuki, Graph neural networks exponentially lose expressive power for node classification. In *ICLR* (2020).

[26] M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li, Simple and deep graph convolutional networks. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13–18 July 2020, Virtual Event*, vol. 119, *Proceedings of Machine Learning Research*, pp. 1725–1735. PMLR (2020). URL http://proceedings.mlr.press/v119/chen20v.html.

[27] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, Graph attention networks. In *ICLR* (2018).

[28] D. Bahdanau, K. Cho, and Y. Bengio, Neural machine translation by jointly learning to align and translate. In eds. Y. Bengio and Y. LeCun, *3rd International Conference on Learning*

Representations, ICLR 2015, San Diego, CA, USA, May 7–9, 2015*, Conference Track Proceedings* (2015). URL http://arxiv.org/abs/1409.0473.

[29] W. Hamilton, Z. Ying, and J. Leskovec, Inductive representation learning on large graphs. In *Proceedings of the Advances in Neural Information Processing Systems,* pp. 1024–1034 (2017).

[30] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, Gated graph sequence neural networks. In *ICLR* (2016).

[31] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, An end-to-end deep learning architecture for graph classification. In *AAAI Conference on Artificial Intelligence* (2018).

[32] M. Niepert, M. Ahmed, and K. Kutzkov, Learning convolutional neural networks for graphs. In *International conference on machine learning*, pp. 2014–2023 (2016).

[33] N. Navarin, D. V. Tran, and A. Sperduti, Learning kernel-based embeddings in graph neural networks. In eds. G. D. Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, *ECAI 2020 - 24th European Conference on Artificial Intelligence,* 29 August–8 September 2020*, Santiago de Compostela,* Spain, August 29–September 8, 2020 — *Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, vol. 325, *Frontiers in Artificial Intelligence and Applications*, pp. 1387–1394. IOS Press (2020). doi: 10.3233/FAIA200243. URL https://doi.org/10.3233/FAIA200243.

[34] N. Shervashidze, P. Schweitzer, E. J. v. Leeuwen, K. Mehlhorn, and K. M. Borgwardt, Weisfeiler-lehman graph kernels, *J. Mach. Learn. Res.,* **12**, 2539–2561 (2011).

[35] F. M. Bianchi, D. Grattarola, C. Alippi, and L. Livi, Graph neural networks with convolutional ARMA filters, *arXiv preprint* (2019).

[36] H. Pei, B. Wei, K. C.-C. Chang, Y. Lei, and B. Yang, Geom-GCN: ceometric graph convolutional networks. In *International Conference on Learning Representations* (2019).

[37] J. Atwood and D. Towsley, Diffusion-convolutional neural networks. In *Neural Information Processing Systems (NIPS)*, pp. 1993–2001 (2016).

[38] D. V. Tran, N. Navarin, and A. Sperduti, On filter size in graph convolutional networks. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1534–1541. IEEE (2018).

[39] F. Wu, T. Zhang, A. H. de Souza, C. Fifty, T. Yu, and K. Q. Weinberger, Simplifying graph convolutional networks, *ICML* (2019).

[40] R. Liao, Z. Zhao, R. Urtasun, and R. S. Zemel, LanczosNet: multi-scale deep graph convolutional networks. In *7th International Conference on Learning Representations, ICLR 2019* (2019).

[41] T. Chen, S. Bian, and Y. Sun, Are powerful graph neural nets necessary? A dissection on graph classification, *arXiv preprint arXiv:1905.04579* (2019).

[42] S. Luan, M. Zhao, X.-W. Chang, and D. Precup, Break the ceiling: stronger multiscale deep graph convolutional networks. In *Proceedings of the Advances in Neural Information Processing Systems,* pp. 10945–10955 (2019).

[43] E. Rossi, F. Frasca, B. Chamberlain, D. Eynard, M. Bronstein, and F. Monti, Sign: scalable inception graph neural networks, *arXiv preprint arXiv:2004.11198* (2020).

[44] M. Liu, H. Gao, and S. Ji, Towards deeper graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 338–348 (2020).

[45] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling, Modeling relational data with graph convolutional networks. In eds. A. Gangemi, R. Navigli, M. Vidal, P. Hitzler, R. Troncy, L. Hollink, A. Tordai, and M. Alam, *The Semantic Web — 15th International Conference, ESWC 2018,* Heraklion, Crete, Greece, June 3–7, 2018*, Proceedings*, vol. 10843, *Lecture Notes in Computer Science*, pp. 593–607. Springer (2018). doi: 10.1007/978-3-319-93417-4\38. URL https://doi.org/10.1007/978-3-319-93417-4_38.

[46] M. Simonovsky and N. Komodakis, Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *CVPR* (2017).

[47] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, Convolutional networks on graphs for learning molecular fingerprints. In *Neural Information Processing Systems (NIPS)*, pp. 2215–2223, Montreal, Canada (2015).

[48] R. C. Read and D. G. Corneil, The graph isomorphism disease, *J. Graph Theory*, **1**(4), 339–363 (1977).

[49] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR 2019,* New Orleans, LA, USA, May 6–9 (2019). OpenReview.net (2019). URL https://openreview.net/forum?id=ryGs6iA5Km.

[50] H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman, Provably powerful graph networks. In *Proceedings of the Advances in Neural Information Processing Systems,* vol. 32 (2019).

[51] C. Morris, G. Rattan, and P. Mutzel, Weisfeiler and Leman go sparse: towards scalable higher-order graph embeddings. In *NeurIPS* (2019).

[52] M. Zaheer, S. Kottur, and S. Ravanbhakhsh, Deep sets. In *Neural Information Processing Systems (NIPS)*, pp. 3391–3401 (2017).

[53] N. Navarin, D. V. Tran, and A. Sperduti, Universal readout for graph convolutional neural networks. In *International Joint Conference on Neural Networks, IJCNN 2019,* Budapest, Hungary, July 14–19, 2019, pp. 1–7. IEEE (2019). doi: 10.1109/IJCNN. 2019.8852103. URL https://doi.org/10.1109/IJCNN.2019.8852103.

[54] L. Pasa, N. Navarin, and A. Sperduti, Som-based aggregation for graph convolutional neural networks, *Neural Comput. Appl.,* 2020. URL https://doi.org/10.1007/s00521-020-05484-4.

[55] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, Hierarchical graph representation learning with differentiable pooling. In *Neural Information Processing Systems (NIPS)* (2018).

[56] C. Cangea, Veličković, N. Jovanović T. Kipf, and P. Li'o, Towards Sparse Hierarchical Graph Classifiers. In *NIPS Relational Representation Learning Workshop* (2018).

[57] F. M. Bianchi, D. Grattarola, and C. Alippi, Spectral clustering with graph neural networks for graph pooling. In *International Conference on Machine Learning*, pp. 874–883. PMLR (2020).

[58] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch, Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity, *J. Med. Chem*., **34**(2), 786–797 (1991). ISSN 0022-2623.

[59] C. Helma, R. D. King, S. Kramer, and A. Srinivasan, The predictive toxicology challenge 2000–2001, *Bioinformatics*, **17**(1), 107–108 (2001).

[60] N. Wale, I. A. Watson, and G. Karypis, Comparison of descriptor spaces for chemical compound retrieval and classification, *Knowl. Inf. Syst.,* **14**(3), 347–375 (2008).

[61] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. Vishwanathan, A. J. Smola, and H.-P. Kriegel, Protein function prediction via graph kernels, *Bioinformatics*, **21**(Suppl 1), 47–56 (2005).

[62] P. D. Dobson and A. J. Doig, Distinguishing enzyme structures from non-enzymes without alignments, *J. Mol. Biol*., **330**(4), 771–783 (2003).

[63] A. K. McCallum, K. Nigam, J. Rennie, and K. Seymore, Automating the construction of internet portals with machine learning, *Inf. Retrieval*, **3**(2), 127–163 (2000). ISSN 13864564. doi: 10.1023/A:1009953814988.

[64] C. L. Giles, K. D. Bollacker, and S. Lawrence, CiteSeer: an automatic citation indexing system, *Proceedings of the ACM International Conference on Digital Libraries*, pp. 89–98 (1998).

[65] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad, Collective classification in network data, *AI Mag*., **29**(3), 93–93 (2008).

[66] P. Yanardag and S. Vishwanathan, A structural smoothing framework for robust graph comparison, *Proceedings of the Advances in Neural Information Processing Systems,* **28**, 2134–2142 (2015).

[67] F. Errica, M. Podda, D. Bacciu, and A. Micheli, A fair comparison of graph neural networks for graph classification. In *International Conference on Learning Representations* (2020).

[68] M. Fey and J. E. Lenssen, Fast graph representation learning with PyTorch Geometric. In *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds* (2019).

[69] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, PyTorch: an imperative style, high-performance deep learning library. In eds. H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc. (2019).

[70] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: a system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283 (2016).

[71] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, Deep graph library: a graph-centric, highly-performant package for graph neural networks, *arXiv preprint arXiv:1909.01315* (2019).

[72] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, Mxnet: a flexible and efficient machine learning library for heterogeneous distributed systems, *arXiv preprint arXiv:1512.01274* (2015).

[73] D. Grattarola and C. Alippi, Graph neural networks in tensor flow and keras with spektral, *arXiv preprint arXiv:2006.12138* (2020).