Chapter 9
Graph Neural Networks: Graph Classification

Christopher Morris

Abstract Recently, graph neural networks emerged as the leading machine learning architecture for supervised learning with graph and relational input. This chapter gives an overview of GNNs for graph classification, i.e., GNNs that learn a graph-level output. Since GNNs compute node-level representations, pooling layers, i.e., layers that learn graph-level representations from node-level representations, are crucial components for successful graph classification. Hence, we give a thorough overview of pooling layers. Further, we overview recent research in understanding GNN's limitations for graph classification and progress in overcoming them. Finally, we survey some graph classification applications of GNNs and overview benchmark datasets for empirical evaluation.

9.1 Introduction

Graph-structured data is ubiquitous across application domains ranging from chemo- and bioinformatics (Barabasi and Oltvai, 2004; Stokes et al, 2020) to image (Simonovsky and Komodakis, 2017) and social network analysis (Easley et al, 2012). To develop successful (supervised) machine learning models in these domains, we need techniques to exploit the graph structure's rich information and the feature information within nodes and edges. In recent years, numerous approaches have been proposed for (supervised) machine learning with graphs—most notably, approaches based on *graph kernels* (Kriege et al, 2020) and, more recently, using graph neural networks (GNNs), see (Chami et al, 2020; Wu et al, 2021d) for a general overview. Graph kernels work by predefining a fixed set of features, following a two-step feature extraction and learning task approach. They first compute a vectorial representation of the graph based on predefined features, e.g., small subgraphs, random

Christopher Morris
CERC in Data Science for Real-Time Decision-Making, Polytechnique Montréal, e-mail: chris@christophermorris.info

walks, neighborhood information, or a positive semi-definite kernel matrix reflecting pairwise graph similarities. The resulting features or the kernel matrix are then plugged into a learning algorithm such as a Support Vector Machine. Hence, they rely on human-made feature engineering.

GNNs promise that they possibly offer better adaption to the learning task at hand by learning feature extraction and downstream tasks in an end-to-end fashion. One of the most prominent tasks for GNNs is graph classification or regression, i.e., predicting the class labels or target values of a set of graphs, such as properties of chemical molecules (Wu et al, 2018). Since GNNs learn vectorial representations of nodes, or node-level representations, for successful graph classification, the *pooling layer*, i.e., a layer that learns a graph-level from node-level representations, is crucial. This pooling layer aims to learn, based on the node-level representations, a vectorial representation that captures the graph structure as a whole. Ideally, one wants a graph-level representation that captures local patterns, their interaction, and global patterns. However, the optimal representation should adapt to the given data distribution. What is more, GNNs for graph classification have recently successfully been applied to an extensive range of application areas, the most promising being in pharmaceutical drug research; see (Gaudelet et al, 2020) for a survey. Other important application areas include fields such as material science (Xie and Grossman, 201f8), process engineering (Schweidtmann et al, 2020), and combinatorial optimization (Cappart et al, 2021), some of which we also survey here.

In the following, we give an overview of GNNs for graph classification. Starting from the mid-nineties' classic works, we survey modern works from the current deep learning era, followed by an in-depth review of recent pooling layers.

Before GNNs emerged as the leading architecture for graph classification, research focused on kernel-based algorithms, so-called graph kernels, which work by predefining a set of features. Starting from the early 2000s, researchers proposed a plethora of graph kernels, based on graph features such as shortest-paths (Borgwardt et al, 2005), random walks (Kang et al, 2012; Sugiyama and Borgwardt, 2015; Zhang et al, 2018i), local neighborhood information (Shervashidze et al, 2011a; Costa and De Grave, 2010; Morris et al, 2017, 2020b), and matchings (Fröhlich et al, 2005; Woźnica et al, 2010; Kriege and Mutzel, 2012; Johansson and Dubhashi, 2015; Kriege et al, 2016; Nikolentzos et al, 2017); see (Kriege et al, 2020; Borgwardt et al, 2020) for thorough surveys. For a thorough survey on GNNs, e.g., see (Hamilton et al, 2017b; Wu et al, 2021d; Chami et al, 2020).

9.2 Graph neural networks for graph classification: Classic works and modern architectures

In the following, we survey classic and modern works of GNNs for graph classification. GNNs layers for graph classification date back to at least the mid-nineties in chemoinformatics. For example, Kireev (1995) derived GNN-like neural architectures to predict chemical molecule properties. The work of (Merkwirth and

Lengauer, 2005) had a similar aim. Gori et al (2005); Scarselli et al (2008) proposed the original GNN architecture, introducing the general formulation that was later reintroduced and refined in (Gilmer et al, 2017) by deriving the general *message-passing* formulation, most modern GNN architectures can be expressed in, see Section 9.2.1.

We divide our overview of modern GNN layers for graph classification into *spatial approaches*, i.e., ones that are purely based on the graph structure by aggregating local information around each node, and *spectral approaches*, i.e., ones that rely on extracting information from the graph's spectrum. Although this division is somewhat arbitrary, we stick to it due to historical reasons. Due to the large body of different GNN layers, we cannot offer a complete survey but focus on representative and influential works.

9.2.1 Spatial approaches

One of the earliest *modern*, spatial GNN architectures for graph classification was presented in (Duvenaud et al, 2015b), focusing on the prediction of chemical molecules' properties. Specifically, the authors propose to design a differentiable variant of the well-known Extended Connectivity Fingerprint (ECFP) (Rogers and Hahn, 2010) from chemoinformatics, which works similar to the computation of the WL feature vector. For the computation of their GNN layer, denoted *Neural Graph Fingerprints*, Duvenaud et al (2015b) first initialize the feature vector $\mathbf{f}^0(v)$ of each node v with features of the corresponding atom, e.g., a one hot-encoding representing the atom type. In each iteration or layer t, they compute a feature representation $\mathbf{f}^t(v)$ for node v as

$$\mathbf{f}^t(v) = \mathbf{f}^{t-1}(v) + \sum_{w \in N(v)} \mathbf{f}^{t-1}(w),$$

followed by the application of a one-layer perceptron. Here, $N(v)$ denotes the *neighborhood* of node v, i.e., $N(v) = \{w \in \mathscr{V} \mid (v,w) \in \mathscr{E}\}$. Since the ECFP usually computes sparse feature vectors for small molecules, they apply a linear layer followed by a softmax function, i.e.,

$$\mathbf{f}^{t(v)} = \text{softmax}(\mathbf{f}^t(v) \cdot H^t),$$

which they interpret as a sparsification layer, where H^t is the parameter matrix of the linear layer. The final pooled graph-level representation is computed by summing over all layers' features, and the resulting feature is fed into an MLP for the downstream regression and classification. The above GNN layer is compared to the ECFP on molecular regression datasets showing good performance.

Dai et al (2016) introducted a simple GNN layer inspired by mean-field inference. Concretely, given a graph $\mathscr{G}$, the feature $\mathbf{f}^t(v)$ for node v at layer t is computed as

$$\mathbf{f}^t(v) = \sigma(\mathbf{f}^{t-1}(v) \cdot W_1 + \sum_{w \in N(v)} \mathbf{f}^{t-1}(w) \cdot W_2), \tag{9.1}$$

where W_1 and W_2 are parameter matrices in $\mathbb{R}^{d \times d}$, which are shared across layers, and $\sigma(\cdot)$ is a component-wise non-linearity. The above layer is evaluated on standard, small-scale benchmark datasets (Kersting et al, 2016) showing good performance, similar to classical kernel approaches. Lei et al (2017a) proposed a similar layer and showed a connection to kernel approaches by deriving the corresponding kernel space of the learned graph embeddings.

To explicitly support edge labels, e.g., chemical bonds, Simonovsky and Komodakis (2017) introduced *Edge-Conditioned Convolution*, where a feature for node v is represented as

$$\mathbf{f}^t(v) = \frac{1}{|N(v)|} \sum_{w \in N(v)} F^l(l(w,v), W(l)) \cdot \mathbf{f}^{t-1}(w) + \mathbf{b}^l.$$

Here $l(w,v)$ is the feature (or label) of the edge shared by the nodes v and w. Moreover, $F^l \colon \mathbb{R}^s \to \mathbb{R}^{d_t \times d_{t-1}}$ is a function, where s denotes the number of components of the edge features and d_t and d_{t-1} denotes the number of components of the features of layer t and $(t-1)$, respectively, mapping the edge feature to a matrix in $\mathbb{R}^{d_t \times d_{t-1}}$. Further, the function F^l is parameterized by the matrix W, conditioned on the edge feature l. Finally, $\mathbf{b}^l$ is a bias term, again conditioned on the edge feature l. The above layer is applied to graph classification tasks on small-scale, standard benchmark datasets (Kersting et al, 2016), and point cloud data from the computer vision.

Building on (Scarselli et al, 2008), Gilmer et al (2017) introduced a general *message-passing* framework, unifying most of the proposed GNN architectures sofar. Specifically, Gilmer et al (2017) replaced the inner sum defined over the neighborhood in the above equations by a general permutation-invariant, differentiable function, e.g., a neural network, and substituted the outer sum over the previous and the neighborhood feature representation, e.g., by a column-wise vector concatenation or LSTM-style update step. Thus, in full generality a new feature $\mathbf{f}^t(v)$ is computed as

$$f^{W_1}_{\text{merge}}\Big(\mathbf{f}^{t-1}(v), f^{W_2}_{\text{aggr}}\big(\{\!\{\mathbf{f}^{t-1}(w) \mid w \in N(v)\}\!\}\big)\Big), \tag{9.2}$$

where $f^{W_1}_{\text{aggr}}$ aggregates over the multi-set of neighborhood features and $f^{W_2}_{\text{merge}}$ merges the node's representation from step $(t-1)$ with the computed neighborhood features. Moreover, it is straighfoward to include edge features as well, e.g., by learning a combined feature representation of the node itself, the neighboring node, and the corresponding edge feature. Gilmer et al (2017) employed the above architecture for regression tasks from quantum chemistry, showing promising performance for regression targets computed by expensive numerical simulations (namely, DFT) (Wu et al, 2018; Ramakrishnan et al, 2014).

Concurrently with (Morris et al, 2020b), Xu et al (2019d) investigated the limits of currently used GNN architectures, showing that their expressiveness is bounded by the WL algorithm, a simple heuristic for the graph isomorphism problem. Specifically, they showed that there does not exist a GNN architecture that can distinguish non-isomorphic graphs that the former algorithm cannot. On the positive side, they proposed the *Graph Isomorphism Network* (GIN) layer and showed that there exists a parameter initialization such that it is as expressive as the WL algorithm. Formally, given a graph $\mathscr{G}$ the feature of node v at layer t is computed as

$$\mathbf{f}^t(v) = \mathrm{MLP}\Big((1+\varepsilon)\cdot\mathbf{f}^{t-1}(v) + \sum_{w\in N(v)} \mathbf{f}^{t-1}(w)\Big), \tag{9.3}$$

where MLP is a standard multi-layer perceptron, and ε is a learnable scalar value. Xu et al (2019d) used standard sum pooling, see below, and achieved good results on standard benchmark datasets compared to other standard GNN layers and kernel approaches Morris et al (2020a).

Xu et al (2018a) investigated how to combine local information at different distances from the target node. Concretely, they investigated different architectural design choices for achieving this, e.g., concatenation, max pooling, and LSTM-style attention, showing mild performance improvements on standard benchmark datasets. Moreover, they drew some connection to random-walk distributions.

Niepert et al (2016) studied neural architectures for graph classification by extracting local patterns from graphs. Starting from each vertex, the approach explores the vertex's k-hop neighborhood, e.g., by using a breadth-first strategy. Using a labeling algorithm, e.g., a centrality index, the vertices in this neighborhood are ordered to transform into a fixed-size vector. Afterwards, a CNN-like neural network followed by an MLP is used to perform the final graph classification. The approach is compared to graph kernel approaches on standard, small-scale benchmark datasets (Kersting et al, 2016) showing promising performance.

Corso et al (2020) investigated the effect and limits of neighborhood aggregation functions. They devised aggregation schemes based on multiple aggregators, e.g., sum, mean, minimum, maximum, and standard deviation, together with so-called *degree scalar*, which combat negative effects due to a different number of neighbors between nodes. Specifically, they introduced the scalar

$$S(d,\alpha) = \left(\frac{\log(d+1)}{\delta}\right)^{\alpha},\ d>0, \alpha\in[-1,1],$$

where

$$\delta = \frac{1}{|\mathrm{train}|}\sum_{i\in\mathrm{train}} \log(d_i+1),$$

and α is a variable parameter. Here, the set train contains all nodes i in the training set and d_i denotes its degree, resulting in the aggregation function

$$\bigoplus = \underbrace{\begin{bmatrix} I \\ S(D, \alpha = 1) \\ S(D, \alpha = -1) \end{bmatrix}}_{\text{scalers}} \otimes \underbrace{\begin{bmatrix} \mu \\ \sigma \\ \max \\ \min \end{bmatrix}}_{\text{aggregators}} .$$

where $\otimes$ denotes the tensor product. The authors report promising performance over standard aggregation functions on a wide range of standard benchmark datasets, improving over some standard GNN layers.

Vignac et al (2020b) extended the expressivity of GNNs, see also Section 9.4, by using unique node identifiers, generalizing the message-passing scheme proposed by (Gilmer et al, 2017), see Equation (9.2), by computing and passing matrix features instead of vector features. Formally, each node i maintains a matrix U_i in $\mathbb{R}^{n \times c}$, denoted *local context*, where the j-th row contains the vectorial representation of node j of node i. At initialization, each local context U_i is set to $\mathbf{1}$ in $\mathbb{R}^{n \times 1}$, where n denotes the number of nodes in the given graph. Now at each layer l, similar to the above message-passing framework, the local context is updated as

$$U_i^{(l+1)} = u^{(l)}\left(U_i^{(l)}, \tilde{U}_i^{(l)}\right) \in \mathbb{R}^{n \times c_{l+1}} \quad \text{with} \quad \tilde{U}_i^{(l)} = \phi\left(\left\{m^{(l)}(U_i^{(l)}, U_j^{(l)}, y_{ij})\right\}_{j \in N(i)}\right),$$

where $u^{(l)}, m^{(l)}$, and ϕ are update, message, and aggregation functions, respectively, to compute the updated local context, and y_{ij} denotes the edge features shared by node i and j. Moreover, the authors study the expressive power, showing that, in principle, the above layer can distinguish any non-isomorphic pair of graphs and propose more scalable alternative variants of the above architecture. Finally, promising results on standard benchmark datasets are reported.

9.2.2 Spectral approaches

Spectral approaches apply a convolution operator in the spectral domain of the graph's Laplacian matrix, either by directly computing the former's eigendecomposition or by relying on spectral graph theory, see (Chami et al, 2020; Wang et al, 2018a) for more details. Moreover, they have a solid mathematical foundation stemming from signal processing, see, e.g., (Sandryhaila and Moura, 2013; Shuman et al, 2013).

Formally, let $\mathscr{G}$ be an undirected graph on n nodes with adjacency matrix A, then the *graph Laplacian*

$$L = \mathbf{I} - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

of the graph $\mathscr{G}$, where D is the diagonal matrix of node degrees, i.e., $D_{i,i} = \sum_j (A_{i,j})$. Since the graph Laplacian is positive semi-definite, we can factor it as

$$L = U \Lambda U^{\top},$$

where $U = [\mathbf{u_1}, \ldots, \mathbf{u_n}]$ in $\mathbb{R}^{n \times n}$ denotes the matrix of eigenvectors, sorted according to their eigenvalues. Further, the matrix Λ is a diagonal matrix with $\Lambda_{i,i} = \lambda_i$, where λ_i denotes the ith eigenvalue. Let $\mathbf{x}$ in $\mathbb{R}^n$ be a *graph signal*, i.e., a node feature, then the *graph Fourier transform* and its *inverse* for $\mathbf{x}$ is

$$F(\mathbf{x}) = U^{\top}\mathbf{x} \quad \text{and} \quad F^{-1}(\hat{\mathbf{x}}) = U\mathbf{x},$$

respectively, where $\hat{\mathbf{x}} = F(\mathbf{x})$. Hence, formally, the graph Fourier transform is an orthonormal (linear) transform to the space spanned by the basis of the eigenvectors in U; consequently, each element $\mathbf{x} = \sum_i \hat{\mathbf{x}}_i \cdot \mathbf{u}_i$.

Based on this observation, spectrum-based methods generalize convolution (e.g., on grids) to graphs. Thereto, they learn a *convolution filter* g. Formally, this can be expressed as follows:

$$\mathbf{x} * g = U(U^{\top}\mathbf{x} \odot U^{\top}g) = U \cdot \text{diag}(U^{\top}g) \cdot U^{\top}\mathbf{x},$$

where the operator $\cdot$ denotes the elementwise product. If we set $g_\theta = \text{diag}(U^{\mathsf{T}}g)$, the above can be expressed as

$$\mathbf{x} * g_\theta = U g_\theta U^{\top}\mathbf{x}.$$

Then most spectral approaches differ in their implementation of the operator g_θ.

For example, *Spectral Convolutional Neural Networks* (Bruna et al, 2014) set $g_\theta = \Theta^t_{i,j}$, which is a set of learnable parameters. Based on this, they proposed the following spectral GNN layer:

$$H^t_{\cdot,j} = \sigma\left(\sum_{i=1}^{t} U \Theta^t_{i,j} U^{\top} H^{t-1}_{\cdot,i}\right),$$

for j in $\{1, 2, \ldots, t\}$. Here, t is the layer index, H^{t-1} in $\mathbb{R}^{n \times (t-1)}$ is the graph signal, where $H^0 = X$, i.e., the given graph features, and $\Theta^t_{i,j}$ is a diagonal parameter matrix. However, the above layer suffers from a number of drawbacks: The bases of the eigenvectors is not permution invariant, the layer cannot be applied to a graph with a different structure, and the computation of the eigendecomposition is cubic in the number of nodes. Hence, Henaff et al (2015) proposed more scalable variants of the above layer by building on a smoothness notion in the spectral domain, which reduces the numbers of parameters and acts as a regulizer.

To further make the above layer more scalable, Defferrard et al (2016) introduced *Chebyshev Spectral CNNs*, which approximates g_θ by a Chebyshev expansion (Hammond et al, 2011). Namely, they express

$$g_\theta = \sum_{i=0}^{K} \theta_i T_i(\hat{\Lambda}),$$

where $\hat{\Lambda} = 2\Lambda/\lambda_{\max} - \mathbf{I}$, and $\lambda_{\max}$ denotes the largest eigenvalue of the normalized Laplacian $\hat{\Lambda}$. The normalization ensures that the eigenvalues of the Laplacian are in the $[-1,1]$ real interval, which is required by Chebyshev polynomials. Here, T_i denotes the ith Chebyshev polynomial with $T_1(x) = x$. Alternatively, Levie et al (2019) used Caley polynomials, and show that Chebyshev Spectral CNNs are a special case.

Kipf and Welling (2017b) proposed to make Chebyshev Spectral CNNs more scalable by setting

$$\mathbf{x} * g_\theta = \theta_0 \mathbf{x} - \theta_1 D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \mathbf{x}.$$

Further, they improved the generalization ability of the resulting layer by setting $\theta = \theta_0 = -\theta_1$, resulting in

$$\mathbf{x} * g_\theta = \theta(\mathbf{I} + D^{-\frac{1}{2}} A D^{-\frac{1}{2}})\mathbf{x}.$$

In fact, the above layer can be understood as a spatial GNN, i.e., it is equivalent to computing a feature

$$\mathbf{f}^t(v) = \sigma\left(\sum_{w \in N(v) \cup v} \frac{1}{\sqrt{d_v d_w}} \mathbf{f}^{t-1}(w) \cdot W\right),$$

for node v in the given graph $\mathscr{G}$, where d_v and d_w denote the degrees of node v and w, respectively. Although the above layer was originally proposed for semi-supervised node classification, it is now one of the most widely used ones and has been applied for tasks such as matrix completion (van den Berg et al, 2018), link prediction (Schlichtkrull et al, 2018), and also as a baseline for graph classification (Ying et al, 2018c).

9.3 Pooling layers: Learning graph-level outputs from node-level outputs

Since GNNs learn vectorial node representations, using them for graph classification requires a pooling layer, enabling going from node to graph-level output. Formally, a pooling layer is a parameterized function that maps a multiset of vectors, i.e., learned node-level representations, to a single vector, i.e., the graph-level representation. Arguably, the simplest of such layers are *sum*, *mean*, and *min* or *max pooling*. That is, given a graph $\mathscr{G}$ and a multiset

$$M = \{\!\!\{ \mathbf{f}(v) \in \mathbb{R}^d \mid v \in \mathscr{V} \}\!\!\}$$

of node-level representations of nodes in the graph $\mathscr{G}$, sum pooling computes

$$f_{\text{pool}}(\mathscr{G}) = \sum_{\mathbf{f}(v) \in M} \mathbf{f}(v),$$

while mean, min, max pooling take the (component-wise) average, minimum, maximum over the elements in M, respectively. These four simple pooling layers are still used in many published GNN architectures, e.g., see (Duvenaud et al, 2015b). In fact, recent work (Mesquita et al, 2020) showed that more sophisticated layers, e.g., relying on clustering, see below, do not offer any empirical benefits on many real-world datasets, especially those from the molecular domain.

9.3.1 Attention-based pooling layers

Simple attention-based pooling became popular in recent years due to its easy implementation and scalability compared to more sophisticated alternatives; see below. For example, Gilmer et al (2017), see above, used a *seq2seq* architecture for sets (Vinyals et al, 2016) for pooling purposes in their empirical study. Focusing on pooling for GNNs, Lee et al (2019b) introduced the *SAGPool* layer, short for Self-Attention Graph Pooling method for GNNs, using self-attention. Specifically, they computed a *self-attention score* by multiplying the aggregated features of an arbitrary GNN layer by a matrix Θ_{att} in $\mathbb{R}^{d \times 1}$, where d denotes the number of components of the node features. For example, computing the self-attention score $\mathbf{Z}(v)$ for the simple layer of Equation (9.1) equates to

$$\mathbf{Z}(v) = \sigma \left(\mathbf{f}^{t-1}(v) \cdot W_1 + \sum_{w \in N(v)} \mathbf{f}^{t-1}(w) \cdot W_2 \right) \cdot \Theta_{\text{att}}.$$

The self-attention score $\mathbf{Z}(v)$ is subsequently used to select the top-k nodes in the graph; similarly, to Cangea et al (2018) and (Gao et al, 2018a), see below, omitting the other nodes, effectively pruning nodes from the graph. Similar attention-based techniques are proposed in (Huang et al, 2019).

9.3.2 Cluster-based pooling layers

The idea of cluster-based pooling layers is to coarsen the graph, i.e., merging similar nodes iteratively. One of the earliest uses has been proposed in (Simonovsky and Komodakis, 2017), see above, where the *Graclus* clustering algorithm (Dhillon et al, 2007) is used. However, one has the note that the algorithm is parameter-free, i.e., it does adapt to the learning task at hand.

The arguably most well-known cluster-based pooling layer is *DiffPool* (Ying et al, 2018c). The idea of DiffPool is to iteratively coarsen the graph by learning a soft clustering of nodes, making the otherwise discrete clustering assignment differentiable. Concretely, at layer t, DiffPool learns a soft cluster assigment S in $[0,1]^{n_t \times n_{t+1}}$, where n_t and n_{t+1} are the number of nodes at layer t and $(t+1)$, respectively. Each entry $S_{i,j}$ represents the probablity of node i of layer t being clustered into node j of layer $(t+1)$. In each iteration, the matrix S is computed by

$$S = \text{softmax}(\text{GNN}(A_t, F_t)),$$

where A_t and F_t are the adjacency matrix and the feature matrix of the clustered graph at layer t, and the function GNN is an abitrary GNN layer. Finally, in each layer, the adjacency matrix and the feature matrix are updated as

$$A_{t+1} = S^T A_t S \quad \text{and} \quad F_{t+1} = S^T F_t,$$

respectively.

Empirically, the authors show that the DiffPool layer boosts standard GNN layers' performance, e.g., GraphSage (Hamilton et al, 2017b), on standard, small-scale benchmark datasets (Morris et al, 2020a). The downside of the above layer is the added computational cost. The adjacency matrix becomes dense and real-valued after the first pooling layer, leading to a quadratic cost in the number of nodes for each GNN layer's computation. Moreover, the number of clusters has to be chosen in advance, leading to an increase in hyperparameters.

9.3.3 *Other pooling layers*

Zhang et al (2018g) proposed a pooling layer based on differentiable sorting, denoted *SortPooling*. That is, given the feature matrix F_t of row-wise node features after layer t, SortPooling sorts the rows of F_t in a descending fashion. It truncates the last $n-k$ rows of F_t, or pads with zero rows if $n < k$ for a given graph to unify the graphs' size. Formally, the layer can be written down as

$$F = \text{sort}(F_t) \quad \text{followed by} \quad F_{\text{trunc}} = \text{truncate}(F, k),$$

where the function sort sorts the feature matrix F_t row-wise in a descending fashion, and the functions truncate return the first k of the input matrix. Ties are broken up using the features from previous layers, 1 to $(t-1)$. The resulting tensor F_{trunc} of shape $k \times \sum_{i=1}^{h} d_i$, where d_i denotes the number of features of the ith layer and h the total number of layers, is reshaped into a tensor of size $k(\sum_{i=1}^{h} d_i) \times 1$, row-wise, followed by a standard 1-D convolution with a filter and step size of $\sum_{i=1}^{h} d_i$. Finally, a sequence of max-pooling and 1-D convolutions are applied to identifiy local patterns in the sequence.

Similarly, to combat the high computational cost of some pooling layer, e.g., DiffPool, Cangea et al (2018) introduced a pooling layer dropping $n - \lceil nk \rceil$ nodes of a graph with n nodes in each layer for k in $[0,1)$. The nodes to be dropped are choosen according to a *projection score* against a learnable vector $\mathbf{p}$. Concretly, they compute the score vector

$$\mathbf{y} = \frac{F_t \cdot \mathbf{p}}{\|\mathbf{p}\|} \quad \text{and} \quad I = \text{top-}k(\mathbf{y}, k),$$

where top-k returns top-k indices from a given vector according to $\mathbf{y}$. Finally, the adjacency A_{t+1} is updated by removing rows and columns that are not in I, while the updated feature matrix

$$F_{t+1} = (F_t \odot \tanh(\mathbf{y})).$$

The authors report slightly lower classification accuracies than the DiffPool layer on most employed datasets while being much faster in computation time. A similar approach was presented in (Gao and Ji, 2019).

To derive more expressive graph representations, Murphy et al (2019c,b) propose *relational pooling*. To increase the expressive power of GNN layers, they average over all permutations of a given graph. Formally, let $\mathscr{G}$ be a graph, then a representation

$$\mathbf{f}(\mathscr{G}) = \frac{1}{|\mathscr{V}|} \sum_{\pi \in \Pi} g(A_{\pi,\pi}, [F_\pi, I_{|V|}]) \tag{9.4}$$

is learned, where Π denotes all possible permutations of the rows and columns of the adjacency matrix of $\mathscr{G}$, g is a permutation-invariant function, and $[\cdot,\cdot]$ denotes column-wise matrix concatenation. Moreover. $A_{\pi,\pi}$ permutes the rows and columns of the adjaceny matrix A according to the permutation π in Π, similarly F_π permutes the rows of the feature matrix F. The author showed that the above architecture is more expressive in terms of distinguishing non-isomorphic graphs than the WL algorithm, and proposed sampling-based techniques to speed up the computation.

Bianchi et al (2020) introduced a pooling layer based on spectral clustering (VON-LUXBURG, 2007). Thereto, they train a GNN together with an MLP, followed by a softmax function, against an approximation of a relaxed version of the k-way normalized Min-cut problem (Shi and Malik, 2000). The resulting cluster assignment matrix S is used in the same way as in Section 9.3.2. The authors evaluated their approach on standard, small-scale benchmark datasets showing promising performance, especially over the DiffPool layer. For another pooling layer based on spectral clustering, see (Ma et al, 2019d).

9.4 Limitations of graph neural networks and higher-order layers for graph classification

In the following, we briefly survey the limitations of GNNs and how their expressive power is upper-bounded by the Weisfeiler-Leman method (Weisfeiler and Leman, 1968; Weisfeiler, 1976; Grohe, 2017). Concretely, a recent line of works by Morris et al (2020b); Xu et al (2019d); Maron et al (2019a) connects the power or expressivity of GNNs to that of the WL algorithm. The results show that GNN architectures generally do not have more power to distinguish between non-isomorphic graphs than the WL. That is, for any graph structure that the WL algorithm cannot distinguish, any possible GNN with any possible choices of parameters will also not be able to distinguish it. On the positive side, the second result states that there is a sequence of parameter initializations such that GNNs have the same power in distinguishing non-isomorphic (sub-)graphs as the WL algorithm, see also Equation (9.3). However, the WL algorithm has many short-comings, see (Arvind et al, 2015; Kiefer et al, 2015), e.g., it cannot distinguish between cycles of different lengths, an important property for chemical molecules, and is not able to distinguish between graphs with different triangle counts, an important property of social networks.

To address this, many recent works tried to build provable more expressive GNNs for graph classification. For example, in (Morris et al, 2020b; Maron et al, 2019b, 2018) the authors proposed *higher-order GNN architectures* that have the same expressive power as the *k-dimensional Weisfeiler-Leman algorithm* (k-WL), which is, as k grows, a more expressive generalization of the WL algorithm. In the following, we give an overview of such works.

9.4.1 Overcoming limitations

The first GNN architecture that overcame the limitations of the WL algorithm was proposed in (Morris et al, 2020b). Specifically, they introduced so-called *k-GNNs*, which work by learning features over the set of subgraphs on k nodes instead of vertices by defining a notion of neighborhood between these subgraphs. Formally, for a given k, they consider all k-element subsets $[\mathscr{V}]^k$ over $\mathscr{V}$. Let $s = \{s_1, \ldots, s_k\}$ be a k-set in $[\mathscr{V}]^k$, then they define the *neighborhood* of s as

$$N(s) = \{t \in [\mathscr{V}]^k \mid |s \cap t| = k-1\}\,.$$

The *local neighborhood* $N_L(s)$ consists of all t in $N(s)$ such that (v,w) in $\mathscr{E}$ for the unique $v \in s \setminus t$ and the unique $w \in t \setminus s$. The *global neighborhood* $N_G(s)$ then is defined as $N(s) \setminus N_{\mathrm{L}}(s)$.

Based on this neighborhood definition, one can generalize most GNN layers for vertex embeddings to more expressive subgraph embeddings. Given a graph $\mathscr{G}$, a feature for a subgraph s can be computed as

$$\mathbf{f}_k^t(s) = \sigma\Big(\mathbf{f}_k^{t-1}(s) \cdot W_1^t + \sum_{u \in N_L(s) \cup N_G(s)} \mathbf{f}_k^{t-1}(u) \cdot W_2^t\Big)\,. \tag{9.5}$$

The authors resort to sum over the local neighborhood in the experiments for better scalability and generalization, showing a significant boost over standard GNNs on a quantum chemistry benchmark dataset (Wu et al, 2018; Ramakrishnan et al, 2014).

The latter approach was refined in (Maron et al, 2019a) and (Morris et al, 2019). Specifically, based on (Maron et al, 2018), Maron et al (2019a) derived an architecture based on standard matrix multiplication that has at least the same power as the 3-WL. Morris et al (2019) proposed a variant of the k-WL that, unlike the original algorithm, takes the sparsity of the underlying graph into account. Moreover, they showed that the derived sparse variant is slightly more powerful than the k-WL in distinguishing non-isomorphic graphs and proposed a neural architecture with the same power as the sparse k-WL variant.

An important direction in studying graph representations' expressive power was taken by (Chen et al, 2019f). The authors prove that a graph representation can approximate a function f if and only if it can distinguish all pairs of non-isomorphic graphs $\mathscr{G}$ and $\mathscr{H}$ where $f(\mathscr{G}) \neq f(\mathscr{H})$. With that in mind, they established an equivalence between the set of pairs of graphs a representation can distinguish and the space of functions it can approximate, further introducing a variation of the 2-WL.

Bouritsas et al (2020) enhanced the expressivity of GNNs by annotating node features with subgraph information. Specifically, by fixing a set of predefined, small subgraphs, they annotated each node with their role, formally their automorphism type, in these subgraphs, showing promising performance gains on standard benchmark datasets for graph classification.

Beaini et al (2020) studied how to incorporate directional information into GNNs. Finally, You et al (2021) enhanced GNNs by uniquely coloring central vertices and used two types of message functions to surpass the expressive power of the 1-WL, while Sato et al (2021) and Abboud et al (2020) use random features to achieve the same goal and additionally studied the universality properties of their derived architectures.

9.5 Applications of graph neural networks for graph classification

In the following, we highlight some application areas of GNNs for graph classification, focusing on the molecular domain. One of the most promising applications of GNNs for graph classification is pharmaceutical drug research, see (Gaudelet et al, 2020) for an overview. In this direction, a promosing approach was proposed by (Stokes et al, 2020). They used a form of directed message passing neural networks operating on molecular graphs to identify repurposing candidates for antibiotic development. Moreover, they validated their predictions in vivo, proposing suitable repurposing candidates different from know ones.

Schweidtmann et al (2020) used 2-GNNs, see Equation (9.5), to derive GNN models for predicting three fuel ignition quality indicators such as the derived cetane number, the research octane number,and the motor octane number of oxygenated and non-oxygenated hydrocarbons, indicating that the higher-order layers of Equation (9.5) provide significant gains over standard GNNs in the molecular learning domain.

A general principled GNN for the molecular domain, denoted *DimeNet*, was introduced by (Klicpera et al, 2020). By using an edge-based architecture, they computed a message coefficient between atoms based on their relative positioning in 3D space. Concretely, an incoming message to a node is based on the sender's incoming meassage as well as the distance between the atoms and the angles of their atomic bonds. By using this additional information the authors report significant improvements over state-of-the-art GNN models in molecular property prediction tasks .

9.6 Benchmark Datasets

Since most developments for GNNs are driven empirically, i.e., based on evaluations on standard benchmark datasets, meaningful benchmark datasets are crucial for the development of GNNs in the context of graph classification. Hence, the research community has established several widely used repositories for benchmark datasets for graph classification. Two such repositories are worth being highlighted here. First, the *TUDataset* (Morris et al, 2020a) collection contains over 130 datasets provided at `www.graphlearning.io` of various sizes and various areas such as chemistry, biology, and social networks. Moreover, it provides Python-based data loaders and baseline implementations of standards graph kernel and GNNs. Moreover, the datasets can be easily accessed from well-known GNN implementation frameworks such as *Deep Graph Library* (Wang et al, 2019f), *PyTorch Geometric* (Fey and Lenssen, 2019), or *Spektral* (Grattarola and Alippi, 2020). Secondly, the *OGB* (Weihua Hu, 2020) collections contain many large-scale graph classification benchmark datasets, e.g., from chemistry and code analysis with data loaders, prespecified splits, and evaluation protocols. Finally, Wu et al (2018) also provides many large-scale datasets from chemo- and bioinformatics.

9.7 Summary

We gave an overview of GNNs for graph classification. We surveyed classical and modern works in this area, distinguishing between spatial and spectral approaches. Since GNNs compute node-level representations, pooling layers for learning graph-level representations is crucial for successful graph classification. Hence, we surveyed pooling layers based on attention, clustering, and other approaches to pooling. Moreover, we gave an overview of the limitations of GNNs for graph classification and surveyed architectures to overcome these limitations. Finally, we gave an overview of applications of GNNs and benchmark datasets for their evaluation.

Editor's Notes: The success of using GNNs in classification tasks is owing to advanced representation learning (chapter 2) by expressive power of GNNs (chapter 5). And its performance is limited by the scalability (chapter 6), robustness (chapter 8) and transformation capability (chapter 12) of algorithm. As one of the most prominent tasks, one can always face classification in a variety of GNN topic. For example, node classification helps to evaluate performance of AutoML (chapter17) and self-supervised learning (chapter 18) methods of GNNs, graph classification can be token as subpart of adversarial learning in graph generation (chapter 11). Further, there are many promising applications of GNNs in classification, node or edge based ones like urban intelligence (chapter 27), graph based ones like protein and drug prediction (chapter 25).