arXiv:2110.05292v1 [cs.LG] 11 Oct 2021

Understanding Pooling in Graph Neural Networks

Daniele Grattarola
Università della Svizzera italiana
grattd@usi.ch

Daniele Zambon
Università della Svizzera italiana

Filippo Maria Bianchi
UiT the Arctic University of Norway
NORCE, The Norwegian Research Centre

Cesare Alippi
Università della Svizzera italiana
Politecnico di Milano

Abstract

Inspired by the conventional pooling layers in convolutional neural networks, many recent works in the field of graph machine learning have introduced pooling operators to reduce the size of graphs. The great variety in the literature stems from the many possible strategies for coarsening a graph, which may depend on different assumptions on the graph structure or the specific downstream task. In this paper we propose a formal characterization of graph pooling based on three main operations, called *selection, reduction,* and *connection*, with the goal of unifying the literature under a common framework. Following this formalization, we introduce a taxonomy of pooling operators and categorize more than thirty pooling methods proposed in recent literature. We propose criteria to evaluate the performance of a pooling operator and use them to investigate and contrast the behavior of different classes of the taxonomy on a variety of tasks.

1 Introduction

Similarly to the convolutional and pooling layers in convolutional neural networks (CNNs), graph neural networks (GNNs) are often built by alternating layers that learn a transformation of the node features and pooling layers that reduce the number of nodes. Graph pooling can also be used as an independent operation to produce coarsened representations of given graphs.

While the techniques for learning node representations have been largely studied, and works like those of Gilmer et al. [21] and Battaglia et al. [5] have introduced general frameworks to unify the existing literature, less attention has been devoted to pooling layers. Only a few recent works have attempted to systematically analyze the effect of pooling in GNNs [28, 38] and, notably, a unifying formulation of pooling operators is still missing. In this paper we advance the understanding of graph pooling operators by proposing a universal and modular formalism to study pooling in GNNs and we show what types of operators can be effective in different settings. Specifically:

- We show that graph pooling operators can be seen as the combination of three functions: *selection*, *reduction*, and *connection* (SRC). The selection function groups the nodes of the input graph into subsets called *supernodes*; then, the reduction function aggregates each supernode to form an output node and its attributes; finally, the connection function links the reduced nodes with (possibly attributed) edges and outputs the pooled graph. The process is summarized in Figure 1. We also show that the three SRC functions can be interpreted as node- and graph-embedding operations, and that recent theoretical results on the universality of GNNs [27, 37] can be exploited to define universal approximators for any pooling operator with continuous SRC functions.

- We propose a comprehensive taxonomy of pooling operators based on specific properties of the SRC functions. In particular, we identify four main properties that characterize pooling operators: a)

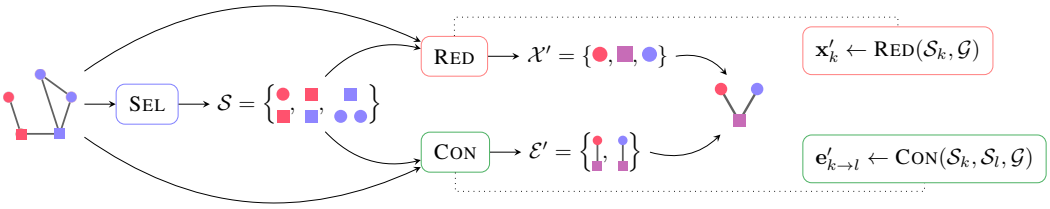


Figure 1: Schematic view of pooling operators: the selection function SEL groups the nodes into supernodes $\mathcal{S}_1, \dots, \mathcal{S}_K$; the reduction function RED maps each supernode $\mathcal{S}_k$ to the attribute of node k in the pooled graph; finally, the connection function CON computes the edges between each pair of new nodes.

whether or not the SRC functions are learned, b) whether their complexity is linear or quadratic in the number of nodes, c) whether they produce graphs with a fixed or variable number of nodes, d) whether they pool the graphs hierarchically or globally.

- We identify three evaluation criteria that allow quantifying how much a pooling operator preserves information related to the node attributes, how much it preserves the original topological structure, and how well it can adapt the coarsened graph in relation to the downstream tasks. Through experiments designed to test how well these criteria are met, we evaluate the performance and analyze the behavior of different classes of pooling operators. We provide guidelines to identify the appropriate pooling methods for a given task based on the taxonomy and the fulfillment of the discussed criteria.

- Finally, we release a standardized and framework-agnostic Python API for implementing graph pooling operators.[1] Such general interfaces are the basis on which popular software libraries for creating GNNs [19, 22] are built, as they offer an easy and flexible way to make novel operators available to the research community and to reproduce results.

2 Pooling in graph neural networks

Among the early uses of graph pooling found in the GNN literature, Bruna et al. [9] mentioned popular graph clustering algorithms [12, 29, 50, 16] to perform pooling in GNN architectures. In particular, the Graclus algorithm [16] was used by Defferrard et al. [15] and later adopted in other works on GNNs [39, 32, 20]. The literature about learning on point clouds also introduced pooling techniques to generalize the typical pooling layers for grids, with most approaches based on voxelization techniques [48, 45, 43, 31]. Many pooling operators have also been proposed based on graph spectral theory [7, 36, 23], or different clustering, sparsification, and decomposition techniques [35, 3, 42, 53].

The current trend (and state of the art) in graph pooling has seen the advent of learnable operators that, much like message-passing layers, can dynamically adapt to a particular task to compute optimal pooling. These include clustering approaches like the DiffPool [55] and MinCut [6] operators, as well as a large variety of methods that learn to keep some nodes and discard the others [24, 11, 28, 30, 44]. Other notable works include those of Diehl [17], Bodnar et al. [8], and a broad group of *global* pooling methods (*cf.* Section 4) that reduce a whole graph to a single vector [33, 58, 51, 41, 13, 2, 54, 4].

This paper studies such a heterogeneous family of operators to highlight their common characteristics and key differences. We start by introducing, in the next section, a definition for graph pooling that encompasses all methods listed above.

Notation We denote an attributed graph with N nodes as a tuple $\mathcal{G} = (\mathcal{X}, \mathcal{E})$ where $\mathcal{X} = \{(i, \mathbf{x}_i)\}_{i=1:N}$ is the node set, with $\mathbf{x}_i \in \mathfrak{X}$ the attribute of the i-th node, and $\mathcal{E} = \big\{\big((i,j), \mathbf{e}_{ij}\big)\big\}_{i,j\in 1:N}$ is the edge set, with $\mathbf{e}_{ij} \in \mathfrak{E}$ the attribute associated with the edge between nodes i and j. With a little abuse of notation, in the following we identify node $(i, \mathbf{x}_i)$ only with its attribute $\mathbf{x}_i$; similarly for edges. Usually, domains $\mathfrak{X}$ and $\mathfrak{E}$ of the node and edge attributes are real vector spaces, namely, $\mathfrak{X} = \mathbb{R}^F$ and $\mathfrak{E} = \mathbb{R}^H$ for $F, H \in \mathbb{N}$. Without loss of generality, non-attributed graphs can still fit in this formalism by considering surrogate attributes, such as a

[1] `https://github.com/danielegrattarola/SRC`

constant value or some node-specific property (*e.g.*, the degree). It is also practical to represent the graph with an adjacency matrix $\mathbf{A} \in \{0,1\}^{N \times N}$ and a node attribute matrix $\mathbf{X} \in \mathbb{R}^{N \times F}$. In this paper, we consider undirected graphs since the literature mostly focuses on them.

3 Select, reduce, connect

Let a graph pooling operator be loosely defined as any function POOL that maps a graph $\mathcal{G}$ to a new pooled graph $\mathcal{G}' = (\mathcal{X}', \mathcal{E}')$, with the generic goal of reducing the number of nodes from N to $K < N$. To facilitate our study of graph pooling methods, it is useful to isolate the main operations that all methods must perform, regardless of their specific implementation. We identify three such operations: selection, reduction and connection (SRC); see Figure 1. With selection, the operator computes K subsets of nodes, each associated with one node of the output $\mathcal{G}'$; we refer to them as *supernodes*. With reduction, the operator aggregates the node attributes in each supernode to obtain the node attributes of $\mathcal{G}'$. Finally, the connection step computes edges among the K reduced nodes.

The SRC operations allow us to easily describe pooling methods, as done in Table 1. Accordingly, we define a pooling operator as any function $\text{POOL} : \mathcal{G} \mapsto \mathcal{G}' = (\mathcal{X}', \mathcal{E}')$ written as the composition of

$$\underbrace{\mathcal{S} = \{\mathcal{S}_k\}_{k=1:K} = \text{SEL}(\mathcal{G});}_{\text{Selection}} \quad \underbrace{\mathcal{X}' = \{\text{RED}(\mathcal{G}, \mathcal{S}_k)\}_{k=1:K};}_{\text{Reduction}} \quad \underbrace{\mathcal{E}' = \{\text{CON}(\mathcal{G}, \mathcal{S}_k, \mathcal{S}_l)\}_{k,l=1:K}.}_{\text{Connection}} \tag{1}$$

Different pooling operators are determined by the specific implementations of SEL, RED and CON.

Select The selection function SEL maps the nodes of the input graph to the nodes of the pooled one. The role of SEL is crucial as it determines the number of nodes in the output graph and what information from the input will be carried over to the output. A selection consists of assigning the N input nodes to K sets $\mathcal{S}_1, \dots, \mathcal{S}_K \subseteq \mathcal{X}$, called *supernodes*: $\text{SEL} : \mathcal{G} \mapsto \mathcal{S} = \{\mathcal{S}_1, \dots, \mathcal{S}_K\}$. Each supernode is a set $\mathcal{S}_k = \{(\mathbf{x}_i, \mathbf{s}_i) \mid \mathbf{x}_i \in \mathcal{X}, \mathbf{s}_i \in \mathbb{R}, \mathbf{s}_i > 0\}$, whose element $(\mathbf{x}_i, \mathbf{s}_i)$ indicates that node i of the input graph impacts on the k-th node of the pooled graph. The value $\mathbf{s}_i$ is a membership score of node i with respect to supernode k, *i.e.*, how much node i contributes to $\mathcal{S}_k$. In general, a node can be assigned to zero, one, or multiple supernodes, with different scores.

Reduce The reduction function computes the node attributes of graph $\mathcal{G}'$ by aggregating the node attributes of $\mathcal{G}$ selected in each supernode $\mathcal{S}_k$. A reduction consists of applying a function RED to each supernode $\mathcal{S}_k$ to produce the k-th node attribute $\mathbf{x}'_k$ of $\mathcal{G}'$: $\text{RED} : \mathcal{G}, \mathcal{S}_k \mapsto \mathbf{x}'_k \in \mathfrak{X}$.

Connect The connection function determines, for each pair of supernodes $\mathcal{S}_k, \mathcal{S}_l$, the presence or absence of an edge between the corresponding nodes k and l in the pooled graph. The function also computes the attributes to be assigned to new edges and reads: $\text{CON} : \mathcal{G}, \mathcal{S}_k, \mathcal{S}_l \mapsto \mathbf{e}'_{kl} \in \mathfrak{E}$. We assume that the space of edge attributes contains a null attribute encoding the absence of an edge and that the edge set of a graph only contains non-null edges.

The reason why both RED and CON are defined as functions of graph $\mathcal{G}$ is that their output can depend on the full topology of the input graph in non-trivial ways. For example, pooling methods based on the graph spectrum often connect the nodes of $\mathcal{G}'$ based on the whole structure of $\mathcal{G}$. However, we notice that many pooling operators implement RED and CON as functions of the supernodes only.

Given our definition of pooling operators as a combination of the SRC functions, we show in Table 1 how to express several pooling methods proposed in recent literature under the SRC formalism. We observe that the selection is commonly computed as a matrix $\mathbf{S} \in \mathbb{R}^{N \times K}$, where $\mathbf{S}_{ik}$ indicates the membership score of node i to supernode k, and $\mathbf{S}_{ik} = 0$ means that node i is not assigned to supernode k.

SRC as embedding operations Since the three SRC functions are essentially node- and graph-embedding operations, we can rely on well-established theory [46] to study the expressive power of pooling operators when formulated under the SRC framework.

Consider a graph space defined on compact node and edge attribute sets $\mathfrak{X}, \mathfrak{E}$, and let $K(\mathcal{G})$ represent the number of nodes of $\mathcal{G}' = \text{POOL}(\mathcal{G})$, where $K(\mathcal{G}) \leq \overline{K}$ for all $\mathcal{G}$ and for some finite $\overline{K} \in \mathbb{N}$. By representing the output of the selection function as a matrix $\mathbf{S} \in \mathbb{R}^{N \times \overline{K}}$, we can then interpret SEL as permutation-equivariant node embedding operation $\mathbf{x}_i \mapsto \mathbf{S}_{i,:}$, from the space of node attributes to the space of supernodes assignments $\mathbb{R}^{\overline{K}}$ where we assumed, without loss of generality, that $\mathbf{S}_{i,k} = 0$ for all $k > K(\mathcal{G})$ (this is necessary to ensure that any number of nodes $K(\mathcal{G})$ can be computed by

Table 1: Pooling methods in the SRC framework. GNN indicates a stack of one or more message-passing layers, MLP is a multi-layer perceptron, $\mathbf{L}$ is the normalized graph Laplacian, β is a regularization vector (see [42]), $\mathbf{D}$ is the degree matrix, $\mathbf{u}_{max}$ is the eigenvector of the Laplacian associated with the largest eigenvalue, $\mathbf{i}$ is a vector of indices, $\mathbf{A}_{\mathbf{i},\mathbf{i}}$ selects the rows and columns of $\mathbf{A}$ according to $\mathbf{i}$.

Method	**Select**	**Reduce**	**Connect**
DiffPool [55]	$\mathbf{S} = \texttt{GNN}_1(\mathbf{A}, \mathbf{X})$ (w/ auxiliary loss)	$\mathbf{X}' = \mathbf{S}^\top \cdot \texttt{GNN}_2(\mathbf{A}, \mathbf{X})$	$\mathbf{A}' = \mathbf{S}^\top \mathbf{A} \mathbf{S}$
MinCut [6]	$\mathbf{S} = \texttt{MLP}(\mathbf{X})$ (w/ auxiliary loss)	$\mathbf{X}' = \mathbf{S}^\top \mathbf{X}$	$\mathbf{A}' = \mathbf{S}^\top \mathbf{A} \mathbf{S}$
NMF [3]	Factorize: $\mathbf{A} = \mathbf{W}\mathbf{H} \rightarrow \mathbf{S} = \mathbf{H}^\top$	$\mathbf{X}' = \mathbf{S}^\top \mathbf{X}$	$\mathbf{A}' = \mathbf{S}^\top \mathbf{A} \mathbf{S}$
LaPool [42]	$\begin{cases} \mathbf{V} = \lVert \mathbf{L}\mathbf{X} \rVert_d ; \\ \mathbf{i} = \{i \mid \forall j \in \mathcal{N}(i) : \mathbf{V}_i > \mathbf{V}_j\} \\ \mathbf{S} = \text{SparseMax}\left(\beta \frac{\mathbf{X}\mathbf{X}_\mathbf{i}^\top}{\lVert \mathbf{X} \rVert \lVert \mathbf{X}_\mathbf{i} \rVert}\right) \end{cases}$	$\mathbf{X}' = \mathbf{S}^\top \mathbf{X}$	$\mathbf{A}' = \mathbf{S}^\top \mathbf{A} \mathbf{S}$
Graclus [16]	$\mathcal{S}_k = \left\{ \mathbf{x}_i, \mathbf{x}_j \mid \arg\max_j \left(\frac{\mathbf{A}_{ij}}{\mathbf{D}_{ii}} + \frac{\mathbf{A}_{ij}}{\mathbf{D}_{jj}} \right) \right\}$	$\mathbf{X}' = \mathbf{S}^\top \mathbf{X}$	METIS [26]
NDP [7]	$\mathbf{i} = \{i \mid \mathbf{u}_{max,i} > 0\}$	$\mathbf{X}' = \mathbf{X}_\mathbf{i}$	Kron r. [18]
Top-K [24]	$\mathbf{y} = \frac{\mathbf{X}\mathbf{p}}{\lVert \mathbf{p} \rVert}$; $\mathbf{i} = \text{top}_K(\mathbf{y})$	$\mathbf{X}' = (\mathbf{X} \odot \sigma(\mathbf{y}))_\mathbf{i}$;	$\mathbf{A}' = \mathbf{A}_{\mathbf{i},\mathbf{i}}$
SAGPool [30]	$\mathbf{y} = \texttt{GNN}(\mathbf{A}, \mathbf{X})$; $\mathbf{i} = \text{top}_K(\mathbf{y})$	$\mathbf{X}' = (\mathbf{X} \odot \sigma(\mathbf{y}))_\mathbf{i}$;	$\mathbf{A}' = \mathbf{A}_{\mathbf{i},\mathbf{i}}$

POOL). Now, let $\mathcal{G}_{\mathcal{S}_k}$ indicate an augmentation of $\mathcal{G}$ such that its node features are $\mathbf{X}_{\mathcal{S}_k} = \mathbf{X} \| \mathbf{S}_{:,k}$, where $\|$ indicates concatenation, and similarly $\mathcal{G}_{(\mathcal{S}_k,\mathcal{S}_l)}$ is defined by $\mathbf{X}_{(\mathcal{S}_k,\mathcal{S}_l)} = \mathbf{X} \| \mathbf{S}_{:,k} \| \mathbf{S}_{:,l}$. It is immediate to see that the augmented graphs are an equivalent way of representing the inputs of RED and CON, which can then be seen as graph-embedding operations of the form:

$$\text{RED} : \mathcal{G}_{\mathcal{S}_k} \mapsto \mathbf{x}'_k; \quad \text{CON} : \mathcal{G}_{(\mathcal{S}_k,\mathcal{S}_l)} \mapsto \mathbf{e}'_{kl}. \tag{2}$$

An interesting consequence of this interpretation of SRC is that by implementing SEL as a universal equivariant network [27], and RED and CON as universal invariant ones [37], the resulting operator is a universal approximator for any arbitrary choice of pooling with continuous SEL, RED and CON.

4 Taxonomy of graph pooling

The SRC framework is a general template to describe pooling operators and it allows us to characterize the different families of pooling methods that are found in the literature. We propose the following taxonomy of pooling operators based on four distinguishing characteristics and we show in Table 2 how existing pooling methods fit into this taxonomy.

Trainability A first distinction among pooling operators is whether SEL, RED, and CON are learned end-to-end as part of the overall GNN architecture. In this case, we say that a method is *trainable*, *i.e.*, the operator has parameters which are learned by optimizing a task-driven loss function, while in all other cases we say that methods are *non-trainable*. This distinction is important because, while non-trainable methods are often used as stand-alone algorithms for graph coarsening, trainable methods were specifically designed for GNNs and are a novel research topic of their own.

Generally, non-trainable methods are useful when there is strong prior information about the desired behavior of pooling (*e.g.*, preserving connectivity [16] or filtering out some particular graph frequencies [42]). These prior assumptions are usually grounded on graph-theoretical properties and are useful when few data are available, since they do not increase the overall number of parameters and introduce no additional optimization objectives when training the GNN. A well-known example of non-trainable pooling is the conventional grid pooling of CNNs, which pools spatially localized groups of pixels. On the other hand, trainable methods are more flexible and make fewer assumptions about the desired result. Therefore, they are useful in problems where the best pooling strategy is not known *a priori*. However, note that it is possible to integrate priors about the desired pooling behavior also in trainable methods (*e.g.*, the MinCut [6] operator also optimizes a normalized cut objective to ensure that supernodes are homogeneous). These additional assumptions usually act as a regularization.

Density of the supernodes A second axis of the taxonomy is concerned with the size of supernodes and the consequent cost of computing the selection function. We define the *density* of a pooling operator as the expected value $\mathbb{E}\left[|\mathcal{S}_k|/N\right]$ of the ratio between the cardinality of a supernode $\mathcal{S}_k$ and the number of nodes in $\mathcal{G}$. We say that a method is *dense* if SEL generates supernodes $\mathcal{S}_k$ whose cardinality is $O(N)$, and *sparse* if supernodes have constant cardinality $O(1)$.[2] Fig. 2(a) shows an example of sparse and dense selection.

This distinction is key, since sparse methods require much less computational resources, especially in terms of memory, which is a significant bottleneck even in modern GPUs. This makes them scale better to large graphs. However, as we show in Section 5, sparse selection is a harder operation to learn than dense selection, and may result in unexpected behaviors.

Adaptability of K It is also possible to distinguish pooling methods according to the number of nodes K of the pooled graph. If K is constant and independent from the input graph size, we say that a pooling method is *fixed*. In this case, K is a hyperparameter of the pooling operator and the output graph will always have K nodes. For example, K can be the number of output features of a neural network used to compute cluster assignments [6, 55]. On the other hand, if the number of supernodes is a function $K(\mathcal{G})$ of the input graph we say that the method is *adaptive*. In many cases, $K(\mathcal{G})$ is a function of N (*e.g.* the ratio $N/2$), but $K(\mathcal{G})$ could also depend on the input graph in a more complex way (*e.g.*, [42, 28]).

Adaptive pooling methods can compute graphs that have a size proportional to that of the input. On the other hand, all the coarsened graphs generated by fixed methods will have the same size. This can lead to situations where $K > N$ for some graphs, causing them to be upscaled by pooling, rather than coarsened. Fig. 2(b) compares fixed and adaptive pooling and shows an example (2^{nd} row) where fixed pooling upscales the graph. For data with a wide or skewed distribution of the number of nodes, the values commonly chosen for K in fixed methods, like the average number of nodes in the training set, may cause several small graphs to be upscaled and very big graphs to be excessively shrunk. Therefore, if the relative graph size is important for solving a particular task, adaptive methods should be preferred.

Hierarchy A distinction often found in the literature is that between "regular" and global pooling, which is extremely evident, to the point where global pooling is usually referred to as a separate operation called "readout". Here we show that such a distinction can be formalized with the SRC framework. Specifically, *global pooling* indicates those methods that reduce a graph to a single node, discarding all topological information. A pooling method is global if it is fixed with $K = 1$, *i.e.*, it returns a degenerate single-node graph represented by its attribute. Also, the connection function is a constant map to the empty set. On the other hand, we indicate all other methods as *hierarchical* pooling operators. Fig. 2(a) shows an example of hierarchical and global pooling.

Hierarchical and global pooling operators have different roles and both can be part of the same GNN architecture for graph-level learning. The former provide a multi-resolution representation of the graph from which the GNN can gradually distill high-level properties, while the latter compute graph embeddings to interface with traditional layers operating on vectors.

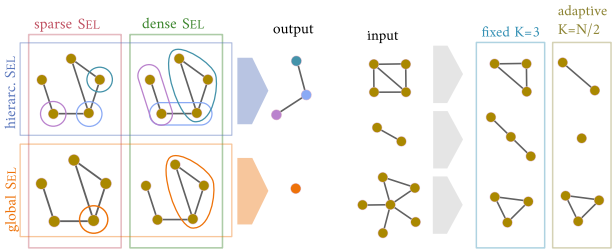
Discussion The main differences among pooling methods are in the selection function, while much less variety is found in the reduction and connection functions. A majority of methods (*e.g.*, [48, 43, 3, 35, 36, 42]) have adaptive K, with a dense and non-trainable selection. Adaptive methods are the most commonly found in the literature, although fixed pooling operators are currently the state of the art [55, 6]. We also note that, to the best of our knowledge, there are no pooling operators that are trainable, dense, and adaptive, which could be an interesting research topic in the near future.

Considering density and adaptability, we see that the memory cost of a pooling operator can range from $O(1)$ (sparse and fixed) to $O(N^2)$ (dense and adaptive). This is especially relevant for trainable methods, which usually need to fit into memory-bound computational units like GPUs and TPUs. Figure 3 shows the maximum number of nodes that can be processed by the trainable methods considered in Section 5, without causing a GPU-out-of-memory exception (details in the appendix). As expected, sparse methods can pool graphs up to four times bigger than dense ones.

[2] Intermediate situations—*e.g.*, $O(\log N)$—are also possible, although here we focus on the two limit cases of $O(N)$ and $O(1)$.

Table 2: Taxonomy of pooling operators. Methods are divided in trainable or non-trainable (**T / nT**), dense or sparse (**D / S**), fixed or adaptive (**F / A**), and hierarchical or global (**H / G**).

Method	**T**	**nT**	**D**	**S**	**F**	**A**	**H**	**G**
DiffPool [55], MinCut [6], StructPool [57]	✓		✓		✓		✓	
Top-K methods [24, 11, 30, 28, 44, 59], Edge Contract. [17]	✓			✓		✓	✓	
Coates and Ng [12], Voxelization-based [48, 45, 43, 31]		✓	✓			✓	✓	
NMF [3], EigenPooling [36], LaPool [42], Clique [35]		✓	✓			✓	✓	
Xie *et al.* [53], MPR [8]		✓	✓			✓	✓	
Graclus [16], NDP [7], Pooling in CNNs		✓		✓		✓	✓	
[33, 49, 41]	✓		✓		✓			✓
[58, 51, 13, 2, 54, 4]		✓	✓		✓			✓
Scarselli *et al.* [47]		✓		✓	✓			✓



(a) (b)

Figure 2: 2(a) Sparse supernodes have a constant cardinality ($|\mathcal{S}_k| = 1$) while dense supernodes scale with the size of the graph. Hierarchical methods reduce the graph gradually, while global methods always return one node. 2(b) Fixed methods return the same number of nodes ($K = 3$) while adaptive methods return graphs of size proportional to the input.

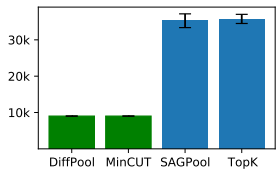


Figure 3: Maximum number of nodes that can be processed by dense (in green) and sparse (in blue) methods.

5 Evaluation

We argue that there is no single general-purpose measure to quantify the performance of a graph pooling algorithm and the quality of a coarsened graph. In this section, we define three evaluation criteria for pooling operators and design experiments to test whether different classes of methods are able to meet them. In particular, we evaluate operators based on their ability to 1) preserve the information content of the node attributes, 2) preserve the topological structure, and 3) preserve the information required to solve various classification tasks. Our goal is to contrast the categories of the proposed taxonomy according to these three criteria. To perform the comparison, we consider the eight hierarchical pooling methods of Table 1 as representatives: MinCut [6] and DiffPool [55] are trainable, dense and fixed; Top-K [24, 11] and SAGPool [30] are trainable, sparse, and adaptive; NMF [3] and LaPool [42] are non-trainable, dense and adaptive; Graclus [16] and NDP [7] are non-trainable, sparse and adaptive. All implementation details are in the appendix.

Preserving node attributes As a first experiment we test the ability of pooling methods to preserve node information. We consider the task of reconstructing the original coordinates of a geometric point cloud from its pooled version. We configure a graph autoencoder to pool the node attributes and then lift them back to the original size using an appropriate lift operator for each method (details in the appendix). Note that this experiment evaluates the quality of the pooling methods in compressing node information, but it does not test their generalization capability since the autoencoder is independently fit on each point could.

Table 3 reports the average and standard deviation of the mean squared error (MSE) obtained by the eight methods on different point clouds from the PyGSP library [14] and the ModelNet40 dataset [52]. Figure 4 contrasts the original point cloud with the points reconstructed from each pooling method while the connectivity is unchanged; the figures for all point clouds are available in the supplementary

Table 3: MSE (values in scale of 10^{-3}) in the autoencoder experiment.[4] The **Rank** row indicates the average ranking of the methods across all datasets.

	Ref. γ	DiffPool	MinCut	NMF	LaPool	TopK	SAGPool	NDP	Graclus
Grid2d	*7.812*	0.010 ±0.005	0.002 ±0.002	**0.000 ±0.000**	0.002 ±0.001	18.86 ±3.923	16.61 ±3.270	**0.000 ±0.000**	0.109 ±0.000
Ring	*4.815*	0.018 ±0.003	0.001 ±0.000	**0.000 ±0.000**	0.052 ±0.046	132.2 ±4.133	148.5 ±30.10	**0.000 ±0.000**	0.600 ±0.000
Bunny	*6.874*	3.901 ±0.275	**0.208 ±0.034**	0.339 ±0.055	0.610 ±0.103	15.32 ±3.557	16.10 ±1.722	0.373 ±0.070	0.332 ±0.043
Airplane	*0.097*	0.094 ±0.022	0.005 ±0.002	0.020 ±0.000	**0.002 ±0.000**	0.096 ±0.028	0.268 ±0.081	0.012 ±0.000	0.009 ±0.000
Car	*0.028*	0.143 ±0.127	0.535 ±0.200	0.016 ±0.001	OOR	0.229 ±0.023	0.204 ±0.029	**0.009 ±0.000**	0.102 ±0.000
Guitar	*0.091*	0.101 ±0.025	0.313 ±0.000	0.007 ±0.000	OOR	0.056 ±0.051	0.060 ±0.044	**0.005 ±0.000**	0.010 ±0.000
Person	*0.013*	0.077 ±0.041	0.301 ±0.000	**0.001 ±0.000**	OOR	0.055 ±0.012	0.062 ±0.033	**0.001 ±0.000**	**0.001 ±0.000**
Rank		5.29	4.29	2.14	5.43	6.14	6.57	**1.86**	3.43

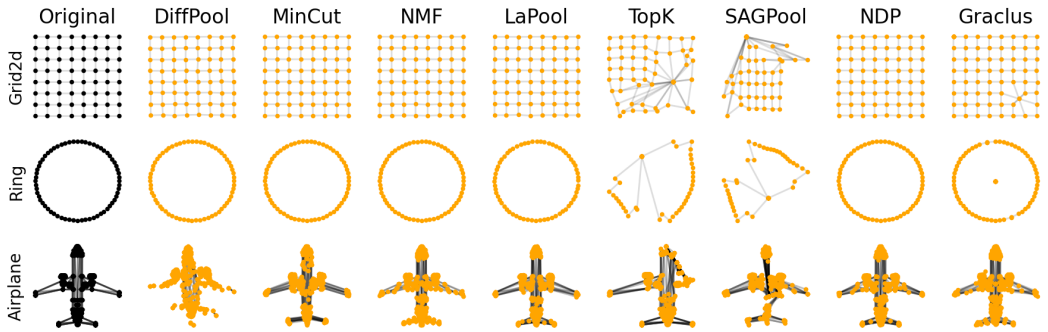


Figure 4: Node attributes (point locations) reconstructed with different operators in the autoencoder experiment.

material. As baseline values for the MSE, we report the mean squared distance between adjacent points: $\gamma = (|\mathcal{E}|\, F)^{-1} \sum_{(i,j)\in\mathcal{E}} |\mathbf{x}_i - \mathbf{x}_j|_2^2$; the intuition behind this baseline is that reconstructed points with a MSE score larger than the reference γ cannot be matched, on average, with the original points. We observe that, as the point clouds grow in size, many operators cannot achieve MSE $< \gamma$ (red entries). The two methods that stand out from Table 3 are the non-trainable NMF and NDP, as confirmed also by their respective average ranks across datasets.

Interpreting the selection operation. Figure 5 depicts a variant of the coarsened graphs where the SEL and CON operations are the same as in Table 1, but the RED function is replaced by

$$\mathbf{X}' = \mathbf{S}^\top \mathbf{X}. \tag{3}$$

This modification is crucial to interpret the SEL operation because most of the pooling methods use message passing layers before the reduction (see the autoencoder architecture details), which makes the node feature space $\mathcal{X}'$ not directly comparable with the original 2 or 3-dimensional input space.

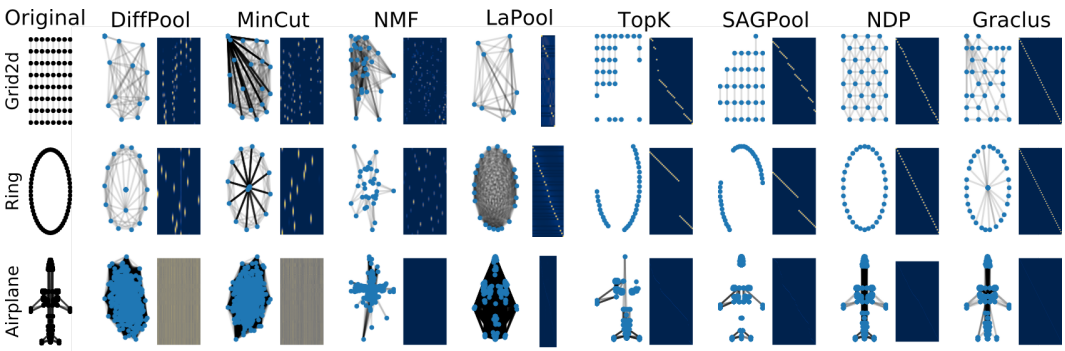


Figure 5: Graphs pooled with different operators in the autoencoder experiment with the modified RED function, and the associated selection matrices $\mathbf{S}$.

Table 4: Average quadratic loss in the spectral similarity experiment.

	DiffPool	MinCut	NMF	LaPool	TopK	SAGPool	NDP	Graclus
Grid2d	**0.002** ±0.000	0.099 ±0.016	0.369 ±0.000	8.486 ±0.000	0.483 ±0.001	0.306 ±0.017	0.068 ±0.000	0.375 ±0.000
Ring	0.001 ±0.000	**0.000** ±0.000	0.050 ±0.000	5.603 ±0.000	0.067 ±0.000	0.034 ±0.001	0.002 ±0.000	0.058 ±0.000
Sensor	**0.010** ±0.000	0.155 ±0.005	1.177 ±0.000	28.99 ±0.000	1.306 ±0.001	0.721 ±0.077	0.486 ±0.000	1.027 ±0.000
Bunny	**0.011** ±0.003	0.272 ±0.013	40.48 ±0.000	$> 10^3$	1.251 ±0.000	0.708 ±0.138	0.156 ±0.000	1.228 ±0.000
Minnes.	**0.000** ±0.000	0.004 ±0.000	7.117 ±0.000	4.030 ±0.000	0.004 ±0.000	0.001 ±0.000	**0.000** ±0.000	0.080 ±0.000
Airfoil	**0.000** ±0.000	0.006 ±0.000	2.604 ±0.000	26.97 ±0.000	0.006 ±0.000	0.003 ±0.000	**0.000** ±0.000	0.048 ±0.000
Rank	**1.17**	2.83	6.33	7.83	5.83	3.67	2.00	5.67

Table 5: Density and median weight of the edges of the coarsened graphs in the spectral similarity experiment.

		Original	DiffPool	MinCut	NMF	LaPool	TopK	SAGPool	NDP	Graclus
Grid2d	**Density**	0.055	0.969	0.969	0.463	0.917	0.084	0.092	0.189	0.103
	Median	1.000	0.216	0.024	0.018	1.445	1.000	1.000	0.500	0.154
Minnes.	**Density**	$9.47{\cdot}10^{-4}$	0.999	0.999	0.010	0.999	0.002	0.002	0.003	0.002
	Median	1.000	0.004	$7.58{\cdot}10^{-4}$	0.014	0.013	1.000	1.000	0.333	0.204
Sensor	**Density**	0.159	0.969	0.969	0.844	0.875	0.273	0.230	0.529	0.227
	Median	0.742	0.463	$2.42{\cdot}10^{-4}$	0.005	6.147	0.765	0.756	0.201	0.103

Conversely, the reduction in (3) gives (weighted) averages of the supernodes with the benefits of maintaining points in the input space and locating them in the supernodes' centres of mass.

Two main patterns emerge. First, we see that the two non-trainable sparse methods (NDP and Graclus) perform a rather uniform node subsampling, in such a way that the reduced node features are good representative of the original input, which may facilitate the reconstruction of the input node features, as confirmed by the low MSE in Table 3. Second, trainable and sparse methods (TopK and SAGPool) tend to cut off entire portions of the graphs, therefore discarding essential node information.

Preserving structure In this experiment we study the structural similarity between the input and coarsened graphs $\mathcal{G}$ and $\mathcal{G}'$, respectively, by comparing the quadratic forms associated with their respective combinatorial Laplacian matrices $\mathbf{L}$ and $\mathbf{L}'$. This evaluation criterion has also been recently studied by Loukas [34], Hermsdorff and Gunderson [23], and Cai et al. [10], and allows us to compare graphs of different sizes. Specifically, we consider the quadratic loss $\mathcal{L}(\mathcal{G}, \mathcal{G}') = \sum_{i=0}^{d} \|\mathbf{X}_{:,i}^\top \mathbf{L} \mathbf{X}_{:,i} - \mathbf{X'}_{:,i}^\top \mathbf{L}' \mathbf{X}'_{:,i}\|$, where $\mathbf{X}$ is an arbitrary graph signal and $\mathbf{X}'$ its reduction. In this experiment, we choose $\mathbf{X}$ to be the concatenation of the first 10 eigenvectors of $\mathbf{L}$ and the node coordinates of $\mathcal{G}$; all columns are ℓ_2-normalized. For trainable methods, we directly minimize the loss as a self-supervised target. Table 4 reports the average loss obtained by the eight operators on different graphs from the PyGSP library, while Figure 6 shows examples of pooled graphs and their spectra. We show the result for Grid2d since it is easier to interpret visually; the figures for all graphs are available in the supplementary material.

Trainable dense methods can generate coarsened graphs with a quadratic loss w.r.t. the original graph lower than their non-trainable or sparse counterparts. Interestingly, from the bottom row of Figure 6 we see that a low quadratic loss does not necessarily imply a good alignment of the spectra. For example, on the regular grid in Figure 6, the excellent spectral alignment achieved by Top-K and SAGPool is not reflected by a low quadratic loss value (0.596 and 0.361 respectively). While in principle this experiment focuses on comparing only SEL and CON, we are also evaluating RED since it affects the loss that depends on $\mathbf{X}'$. This can explain the discrepancy between the loss values and the eigenvalues plots.

Properties of the connection operation. From Figure 6, we see that dense methods (DiffPool, MinCut, NMF, LaPool) yield coarsened graphs that are densely connected. We can make a similar observation for the autoencoder experiment with modified RED operation in Eq. (3), as shown in Figure 5. However, in these dense graphs most of the edge weights are also small. This is quantitatively reported in Table 5, in which we compare the density of edges ($|\mathcal{E}'|/K^2$) and the median edge weight of the coarsened graphs for Grid2d, Minnesota and Sensor. An extended version of this table is reported in the supplementary material.

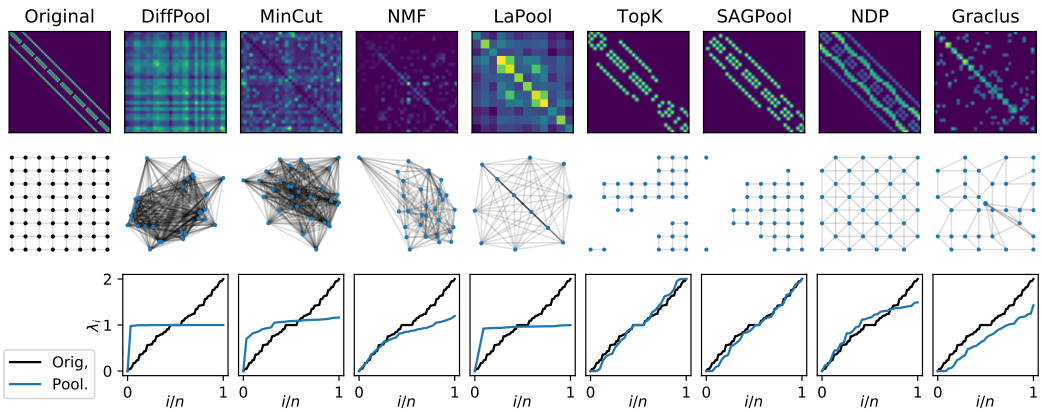


Figure 6: Results on a regular grid when optimizing for spectral similarity. Top: the coarsened adjacency matrices. Middle: the coarsened graphs with modified RED function. Bottom: the eigenvalues of the normalized Laplacian before (black) and after (blue) pooling. The indices of the eigenvalues are rescaled to fill $[0, 1]$.

Table 6: Accuracy on the graph classification benchmarks.[4]

	No-pool	**DiffPool**	**MinCut**	**NMF**	**LaPool**	**TopK**	**SAGPool**	**NDP**	**Graclus**
Colors-3	40.8 ±2.1	55.2 ±1.5	**60.1 ±4.0**	29.7 ±1.7	44.9 ±1.0	26.9 ±4.0	34.4 ±5.2	25.4 ±1.8	29.5 ±2.0
Triangles	93.5 ±0.7	91.3 ±0.2	**95.3 ±0.5**	58.1 ±5.2	88.8 ±0.8	75.2 ±17.3	80.3 ±8.6	75.3 ±1.0	71.4 ±1.7
Proteins	68.8 ±2.8	70.0 ±0.6	**73.8 ±0.8**	68.9 ±3.4	72.9 ±2.0	71.3 ±0.8	73.7 ±0.8	68.4 ±3.4	72.6 ±1.1
Enzymes	83.6 ±2.0	72.4 ±3.9	83.6 ±0.6	32.4 ±8.1	85.0 ±1.2	81.0 ±0.4	68.8 ±18.8	84.8 ±3.2	**85.4 ±4.1**
DD	81.1 ±0.4	75.6 ±1.8	**82.5 ±0.9**	OOR	OOR	80.4 ±0.9	79.0 ±2.7	79.6 ±1.2	78.3 ±2.9
Mutagen.	**78.0 ±1.6**	76.2 ±1.4	73.9 ±1.6	70.3 ±1.6	75.3 ±0.1	75.8 ±1.4	76.9 ±1.4	76.9 ±1.0	74.2 ±0.5
ModelNet	81.0 ±0.5	70.4 ±2.4	75.9 ±1.2	OOR	OOR	74.1 ±3.0	71.9 ±2.6	77.1 ±2.6	**83.9 ±1.9**
Rank		4.43	**2.57**	7.14	4.29	4.71	3.86	4.29	4.29

Preserving task-specific information In our final experiment we consider several benchmarks of graph classification to test the third criterion. A high classification accuracy implies that an operator can selectively preserve information based on the requirements of the task at hand. We consider graph classification problems from the TUDataset [40], the ModelNet10 dataset [52], and the Colors-3 and Triangles datasets introduced by Knyazev et al. [28].

Table 6 reports the average and standard deviation of the classification accuracy on the test set, as well as the average ranking of the operators. We also report as baseline the classification accuracy of a GNN with no pooling (No-pool). We observe that, on the datasets considered here, the operators based on graph spectral properties (MinCut, NDP, Graclus) achieve the highest accuracy. However, we could not find strong evidence that one pooling operator (or even a class of operators) is systematically better than all others. For instance, on Triangles and Colors-3 we see that dense, trainable operators have a consistent advantage. However, the family of sparse and/or non-trainable methods achieves a better performance on Enzymes, Mutagenicity, and the large-scale ModelNet10 dataset. Finally, in datasets such as Mutagenicity, Proteins, and DD the gap in performance is not very large. Table 6 also shows that some of the models with graph pooling operators achieve higher classification accuracy than the no-pool baseline (in green). In Mutagenicity the baseline architecture achieves top performance, suggesting that graph pooling is not always beneficial in certain graph classification tasks. Further discussion can be found in the recent work of Mesquita et al. [38].

[4] "OOR" indicates Out Of Resources, *i.e.*, either we could not fit a batch size of 8 graphs on an Nvidia Titan V GPU or it took more than 24 hours to complete training. Values of 0.000 indicate any value $< 10^{-6}$.

6 Conclusions

In this paper we presented SRC, a unifying formulation of pooling operators in GNNs, that allowed us to organize the vast literature on the subject under a comprehensive taxonomy and implement every pooling operator under a well-defined and modular framework. Based on our framework, we gave pointers towards a possible implementation of a "universal" pooling strategy based on well-known theory about the expressive power of GNNs. We identified three possible evaluation criteria for pooling operators and compared the different classes of the taxonomy on synthetic and real-world benchmarks. Results showed that pooling operators are not all equivalent and, in particular, there is no clear evidence that a particular pooling operator is consistently better than the others according to the evaluation criteria taken into consideration. In our study, we showed the characteristics of different classes of operators and we reported an in-depth analysis of their inner mechanisms.

Overall, we showed that the choice of the best pooling operator, and whether performing graph pooling is necessary at all, highly depends on the problem at hand. A comprehensive evaluation of pooling operators requires considering multiple criteria to highlight all their fundamental properties and, as such, it cannot be limited to measuring the downstream performance on few small-scale benchmark datasets.

Guidelines For the above reasons, we provide guidelines to choose a pooling method in practice, based on the desiderata and the taxonomy:

- To preserve node attributes, especially concerning geometric graphs, non-trainable and sparse methods are suggested since they usually compute a uniform coarsening of the graph;
- To preserve structure, dense and trainable methods are better at minimizing the Laplacian quadratic loss although sparse methods yield a better spectral alignment;
- To preserve task-specific information, which is the most common setting in machine learning, trainable methods have an edge over their counterpart although our conclusion is that there is no better method a priori. Non-trainable sparse methods (Graclus, NDP) have overall better performance across different tasks and are advised as the first choice; however, if the goal is to optimize for a specific objective, then trainable dense methods (MinCut, DiffPool) offer more flexibility and are more easily integrated into GNN architectures.
- We also observe that trainable sparse methods discard entire portions of the graphs and, therefore, they are generally less advisable.

We believe that SRC will be helpful in further studying graph pooling, and that our analysis will guide practitioners in choosing an appropriate pooling method for the application at hand. Our work also leads to a more principled approach in designing and evaluating new pooling operators, and will help the research community to advance the field.

Acknowledgments

This research is funded by the Swiss National Science Foundation project 200021 172671 "ALPS-FORT". We gratefully acknowledge the support of Nvidia Corporation with the donation of the Titan XP GPU used for this work.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] J. Atwood and D. Towsley. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1993–2001, 2016.

[3] D. Bacciu and L. Di Sotto. A non-negative factorization approach to node pooling in graph convolutional neural networks. In *Proceedings of the 18th International Conference of the Italian Association for Artificial Intelligence*. AIIA, 2019.

[4] Y. Bai, H. Ding, Y. Qiao, A. Marinovic, K. Gu, T. Chen, Y. Sun, and W. Wang. Unsupervised inductive graph-level representation learning via graph-graph proximity. *arXiv preprint arXiv:1904.01098*, 2019.

[5] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

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

[7] F. M. Bianchi, D. Grattarola, L. Livi, and C. Alippi. Hierarchical representation learning in graph neural networks with node decimation pooling. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[8] C. Bodnar, C. Cangea, and P. Liò. Deep graph mapper: Seeing graphs through the neural lens. *arXiv preprint arXiv:2002.03864*, 2020.

[9] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[10] C. Cai, D. Wang, and Y. Wang. Graph coarsening with neural networks. *arXiv preprint arXiv:2102.01350*, 2021.

[11] C. Cangea, P. Velićković, N. Jovanović, T. Kipf, and P. Liò. Towards sparse hierarchical graph classifiers. *arXiv preprint arXiv:1811.01287*, 2018.

[12] A. Coates and A. Y. Ng. Selecting receptive fields in deep networks. In *Advances in neural information processing systems*, pages 2528–2536, 2011.

[13] P. Corcoran. Function space pooling for graph convolutional networks. *arXiv preprint arXiv:1905.06259*, 2019.

[14] M. Defferrard, L. Martin, R. Pena, and N. Perraudin. Pygsp: Graph signal processing in python. URL `https://github.com/epfl-lts2/pygsp/`.

[15] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852, 2016.

[16] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence*, 29(11):1944–1957, 2007.

[17] F. Diehl. Edge contraction pooling for graph neural networks. *CoRR*, abs/1905.10990, 2019. URL `http://arxiv.org/abs/1905.10990`.

[18] F. Dorfler and F. Bullo. Kron reduction of graphs with applications to electrical networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 60(1):150–163, Jan 2013. ISSN 1549-8328. doi: 10.1109/TCSI.2012.2215780.

[19] M. Fey and J. E. Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[20] M. Fey, J. E. Lenssen, F. Weichert, and H. Müller. Splinecnn: Fast geometric deep learning with continuous b-spline kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 869–877, 2018.

[21] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

[22] D. Grattarola and C. Alippi. Graph neural networks in tensorflow and keras with spektral. *IEEE Computational Intelligence Magazine*, 2021.

[23] G. B. Hermsdorff and L. M. Gunderson. A unifying framework for spectrum-preserving graph sparsification and coarsening. *arXiv preprint arXiv:1902.09702*, 2019.

[24] S. J. Hongyang Gao. Graph u-net. *Submitted to ICLR*, 2019.

[25] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[26] G. Karypis. Metis: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997.

[27] N. Keriven and G. Peyré. Universal invariant and equivariant graph neural networks. In *Advances in Neural Information Processing Systems 32*, pages 7090–7099. Curran Associates, Inc., 2019. URL `http://papers.nips.cc/paper/8931-universal-invariant-and-equivariant-graph-neural-networks.pdf`.

[28] B. Knyazev, G. W. Taylor, and M. R. Amer. Understanding attention in graph neural networks. *CoRR*, abs/1905.02850, 2019. URL `http://arxiv.org/abs/1905.02850`.

[29] D. Kushnir, M. Galun, and A. Brandt. Fast multiscale clustering and manifold identification. *Pattern Recognition*, 39(10):1876–1891, 2006.

[30] J. Lee, I. Lee, and J. Kang. Self-attention graph pooling. *arXiv preprint arXiv:1904.08082*, 2019.

[31] H. Lei, N. Akhtar, and A. Mian. Octree guided cnn with spherical kernels for 3d point clouds. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9631–9640, 2019.

[32] R. Levie, F. Monti, X. Bresson, and M. M. Bronstein. Cayleynets: Graph convolutional neural networks with complex rational spectral filters. *arXiv preprint arXiv:1705.07664*, 2017.

[33] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[34] A. Loukas. Graph reduction with spectral and cut guarantees. *Journal of Machine Learning Research*, 20(116):1–42, 2019.

[35] E. Luzhnica, B. Day, and P. Lio. Clique pooling for graph classification. *International Conference of Learning Representations (ICLR) – Representation Learning on Graphs and Manifolds workshop*, 2019.

[36] Y. Ma, S. Wang, C. C. Aggarwal, and J. Tang. Graph convolutional networks with eigenpooling. *arXiv preprint arXiv:1904.13107*, 2019.

[37] H. Maron, E. Fetaya, N. Segol, and Y. Lipman. On the universality of invariant networks. In *International Conference on Machine Learning*, pages 4363–4371, 2019.

[38] D. Mesquita, A. Souza, and S. Kaski. Rethinking pooling in graph neural networks. *Advances in Neural Information Processing Systems*, 33, 2020.

[39] F. Monti, D. Boscaini, J. Masci, E. Rodola, J. Svoboda, and M. M. Bronstein. Geometric deep learning on graphs and manifolds using mixture model cnns. In *Proc. CVPR*, volume 1, page 3, 2017.

[40] C. Morris, N. M. Kriege, F. Bause, K. Kersting, P. Mutzel, and M. Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*, 2020.

[41] N. Navarin, D. Van Tran, and A. Sperduti. Universal readout for graph convolutional neural networks. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2019.

[42] E. Noutahi, D. Beani, J. Horwood, and P. Tossou. Towards interpretable sparse graph representation learning with laplacian pooling. *arXiv preprint arXiv:1905.11577*, 2019.

[43] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. In *Advances in neural information processing systems*, pages 5099–5108, 2017.

[44] E. Ranjan, S. Sanyal, and P. P. Talukdar. Asap: Adaptive structure aware pooling for learning hierarchical graph representations. *arXiv preprint arXiv:1911.07979*, 2019.

[45] G. Riegler, A. Osman Ulusoy, and A. Geiger. Octnet: Learning deep 3d representations at high resolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3577–3586, 2017.

[46] R. Sato. A survey on the expressive power of graph neural networks. *arXiv preprint arXiv:2003.04078*, 2020.

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

[48] M. Simonovsky and N. Komodakis. Dynamic edgeconditioned filters in convolutional neural networks on graphs. In *Proc. CVPR*, 2017.

[49] O. Vinyals, S. Bengio, and M. Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015.

[50] U. Von Luxburg. A tutorial on spectral clustering. *Statistics and computing*, 17(4):395–416, 2007.

[51] J. Wu, J. He, and J. Xu. Net: Degree-specific graph neural networks for node and graph classification. *arXiv preprint arXiv:1906.02319*, 2019.

[52] Z. Wu, S. Song, A. Khosla, F. Yu, L. Zhang, X. Tang, and J. Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

[53] Y. Xie, C. Yao, M. Gong, C. Chen, and A. Qin. Graph convolutional networks with multi-level coarsening for graph classification. *Knowledge-Based Systems*, page 105578, 2020.

[54] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[55] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec. Hierarchical graph representation learning with differentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

[56] J. You, Z. Ying, and J. Leskovec. Design space for graph neural networks. *Advances in Neural Information Processing Systems*, 33, 2020.

[57] H. Yuan and S. Ji. Structpool: Structured graph pooling via conditional random fields. In *International Conference on Learning Representations*, 2020. URL `https://openreview.net/forum?id=BJxg_hVtwH`.

[58] M. Zhang, Z. Cui, M. Neumann, and Y. Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of AAAI Conference on Artificial Inteligence*, 2018.

[59] Z. Zhang, J. Bu, M. Ester, J. Zhang, Z. Li, C. Yao, D. Huifen, Z. Yu, and C. Wang. Hierarchical multi-view graph pooling with structure learning. *IEEE Transactions on Knowledge and Data Engineering*, 2021.

A Experimental Details

A.1 Hardware and software

All experiments were run on an NVIDIA Titan V GPU with 12 GB of video memory. All models were implemented in Python 3.7, and all machine learning code was based on TensorFlow 2.4 [1] and Spektral 1.0.6 [22]. The code to reproduce the experiments is available at `https://github.com/danielegrattarola/SRC`.

A.2 Preliminaries

We use the same message-passing model in all of our experiments. The model is inspired by the work of You et al. [56] and has the following form:

$$\mathbf{x}_i' = \mathbf{x}_i \Big\| \sum_{j \in \mathcal{N}(i)} \mathrm{ReLU}\left(\mathrm{BN}\left(\mathbf{W}\mathbf{x}_j + \mathbf{b}\right)\right) \tag{4}$$

where BN indicates batch normalization [25], ReLU is the rectified linear unit, $\mathbf{x}_i \in \mathbb{R}^{d_{in}}$, $\mathbf{x}_i' \in \mathbb{R}^{d_{out}}$, $\mathbf{W} \in \mathbb{R}^{d_{out} \times d_{in}}$ is a trainable matrix, $\mathbf{b} \in \mathbb{R}^{d_{out}}$ is a trainable bias, and $\Big\|$ indicates concatenation. We configure all layers to have $d_{out} = 256$.

We use $\mathrm{GNN}(\mathbf{A}, \mathbf{X})$ to indicate the application of one layer as in Eq. (4) to a graph described by $\mathbf{A}$ and $\mathbf{X}$.

We use $\mathrm{MLP}(\mathbf{X})$ to indicate the application, to the features of each node, of one multi-layer perceptron (MLP) with 2 layers, 256 hidden units, ReLU activation and batch normalization. The number of neurons of each layer is implied by the dimension of the data or, if not specified, is also set to 256 units (*e.g.*, for MLPs used as intermediate blocks).

All architectures and hyperparameters are also inspired by the work You et al. [56]. Specifically, we include a pre-processing and post-processing MLP as first and last blocks of each architecture, respectively. The activation of the last layer of the post-processing MLP is implied by the task.

We use $\textsc{Pool}(\mathbf{A}, \mathbf{X})$ to indicate the application of a generic pooling layer, which is substituted in the architecture with the different operators that we compared in our experiments. Note that, for clarity, we change notation w.r.t. the main text and indicate with $\mathbf{X}_{pool}$, $\mathbf{A}_{pool}$ the output of a pooling layer.

A.3 Preserving node attributes

Architecture The bottleneck of the autoencoder consists of a single pooling layer to compress the graph signal, immediately followed by an *upscaling* layer. The overall architecture is:

$$\begin{aligned}
\mathbf{X}, \mathbf{A} &\leftarrow \mathcal{G} \\
\mathbf{X}_{in} &\leftarrow \mathrm{MLP}_{in}\left(\mathbf{X}\right) \\
\mathbf{X}_{in} &\leftarrow \mathrm{GNN}_{in}\left(\mathbf{A}, \mathbf{X}_{in}\right) \\
\mathbf{X}_{pool} &\leftarrow \textsc{Pool}\left(\mathbf{A}, \mathbf{X}_{in}\right) \\
\mathbf{X}_{up} &\leftarrow \textsc{UpScale}\left(\mathbf{X}_{pool}\right) \\
\mathbf{X}_{out} &\leftarrow \mathrm{GNN}_{out}\left(\mathbf{A}, \mathbf{X}_{up}\right) \\
\mathbf{X}_{out} &\leftarrow \mathrm{MLP}_{out}\left(\mathbf{X}_{out}\right).
\end{aligned}$$

All pooling layers are configured to reduce the graph to $K = \lfloor N/2 \rfloor$ nodes, except for LaPool which determines the number of output nodes autonomously.

The $\textsc{UpScale}$ layer lifts the reduced node features $\mathbf{X}'$ of the coarsened graph $\mathcal{G}'$ back to the original data dimensionality of $\mathbf{X}_{in}$. For almost all methods, the upscaled node features $\mathbf{X}_{up}$ are constructed as $\mathbf{U}\mathbf{X}'$, where $\mathbf{U} = \mathbf{S}^{\mp}$ is the transposed pseudo-inverse of the selection matrix $\mathbf{S} = \textsc{Sel}(\mathbf{X}, \mathbf{A})$. This is the optimal choice when the reduction function is $\mathbf{X}' = \mathbf{S}^\top \mathbf{X}_{in}$. Conversely, for Top-$K$ and SAGPool we need to account also for the scaling factors $\sigma(\mathbf{y})$, so we have $\mathbf{U} = (\sigma(\mathbf{y}) \odot \mathbf{S})^{\mp}$. Finally, even though DiffPool's reduction is $\mathbf{X}' = \mathbf{S}^\top \mathrm{GNN}(\mathbf{A}, \mathbf{X})$, we still employ $\mathbf{U}\mathbf{S}^{\mp}$ as upscaling operator and allow the output layer GNN_{out} to counteract the effects of the GNN.

Table 7: Statistics of graphs used in the autoencoder and spectral similarity experiments.

Graph	Nodes	Edges	Avg. degree
Grid	64	112	3.5
Ring	64	64	2
Bunny	2503	65490	52.35
Airfoil	4253	12289	5.77
Sensor	64	313.7 ± 21.9	9.8 ± 0.8
Airplane	1333	2611	3.91
Car	1920	2372	2.47
Guitar	3125	5508	3.52
Person	3305	9055	5.47

Notice that GNN_{out} takes as input the original adjacency matrix $\mathbf{A}$, so as to focus the experiment on the node features only and suppressing possible interference of the CON function.

Training All models are trained to convergence using Adam to minimize the mean squared error between the input and output features, with learning rate 0.0005 and early stopping on the training loss with a patience of 1000 epochs and a tolerance of 10^{-6}. Each experiment is repeated 3 times.

Data The Grid, Ring, and Bunny graphs are generated using the PyGSP library [14]. The Airplane, Car, Person, and Guitar graphs are taken from the ModelNet40 dataset [52]. We selected one graph randomly from the training set of each category (with a threshold on the number of nodes). The ModelNet40 IDs of the selected graph are: sample 151 for Airplane, sample 75 for Car, sample 38 for Guitar, sample 83 for Person. Details on the size and average degrees of the graphs are reported in Table 7. All datasets that were not generated programmatically are unlicensed.

A.4 Preserving structure

Architecture The architecture for this experiment consists only of a pooling layer, to ensure that the model is actually operating on the original coordinates and eigenvectors without transformations:

$$\begin{aligned} \mathbf{X}, \mathbf{A} &\leftarrow \mathcal{G} \\ \mathbf{X}_{pool} &\leftarrow \text{POOL}\left(\mathbf{A}, \mathbf{X}\right) \end{aligned}$$

All pooling layers are configured to reduce the graph to $K = \lfloor N/2 \rfloor$ nodes, except for LaPool which determines the number of output nodes autonomously.

Training For trainable models, we train them to convergence using Adam to minimize the quadratic loss described in the main text, with learning rate 0.01 and early stopping on the training loss with a patience of 50 epochs and a tolerance of 10^{-6}. Each experiment is repeated 3 times.

Data All graphs are generated using the PyGSP library. In particular, Sensor is a random graph which is generated once per experiment and used for all models. Details on the size and average degrees of the graphs are reported in Table 7. All datasets that were not generated programmatically are unlicensed.

A.5 Preserving task-specific information

Architecture We use the following architecture for all datasets:

$$
\begin{aligned}
\mathbf{X}, \mathbf{A} &\leftarrow \mathcal{G} \\
\mathbf{X} &\leftarrow \text{MLP}_1(\mathbf{X}) \\
\mathbf{X} &\leftarrow \text{GNN}_1(\mathbf{A}, \mathbf{X}) \\
\mathbf{A}_{pool}, \mathbf{X}_{pool} &\leftarrow \text{POOL}(\mathbf{A}, \mathbf{X}) \\
\mathbf{X}_{pool} &\leftarrow \text{GNN}_2(\mathbf{A}_{pool}, \mathbf{X}_{pool}) \\
\mathbf{x}_{out} &\leftarrow \sum_i \mathbf{x}_{pool,i} \\
\mathbf{x}_{out} &\leftarrow \text{MLP}_2(\mathbf{x}_{out})
\end{aligned}
$$

Adaptive pooling layers are configured to reduce the graph to $K = \lfloor N/2 \rfloor$ nodes, except for LaPool which determines the number of output nodes autonomously. Fixed pooling layers are configured to return $K = \left\lfloor \bar{N}/2 \right\rfloor$ nodes, where $\bar{N}$ is the average number of nodes in the training set.

Training All models are trained to convergence using Adam, with a batch size of 16, learning rate 0.0005 and early stopping on the validation loss with a patience of 50 epochs. Each experiment is repeated 3 times.

Data Proteins, Enzymes, Mutagenicity, DD, Colors-3 and Triangles are taken from the TUDataset collection [40]. The ModelNet10 dataset is taken from its original source [52]. All datasets are split randomly according to an 8:1:1 proportion between training, validation and test sets. The only exceptions are Colors-3 and Triangles, for which the data splits are described in [28], and ModelNet10, for which the data splits are given. The TUDatasets are unlicensed and ModelNet10 is provided freely for academic use.

A.6 Memory usage

To produce Figure 3, we generated random Erdős-Rényi graphs with $p = 0.1$ and random features, and gave it as input to the trainable pooling methods (MinCutPool, DiffPool, Top-K and SAGPool). At each forward pass, we increased the number of nodes by 1000 until an out-of-memory exception was raised. We used a sparse tensor to represent the adjacency matrix of the input graphs (so that the cost of loading $\mathbf{A}$ into memory is linear in the number of edges). We repeated the experiments with node features of size $F = 1, 10, 100, 1000$ and found no significant differences in the results.

B Additional results

B.1 Preserving node attributes

An extended version of Figure 4 in the main paper is reported in Figure 7 here. An extended version of Figure 5 in the main paper is reported in Figures 8 and 9. Note that the missing plots for LaPool are due to the Out Of Memory exception as reported in Table 3 in the main paper.

B.2 Preserving structure

An extended version of Figure 6 in the main paper is reported in Figure 10 here. An extended version of Table 5 is reported in Table 8.

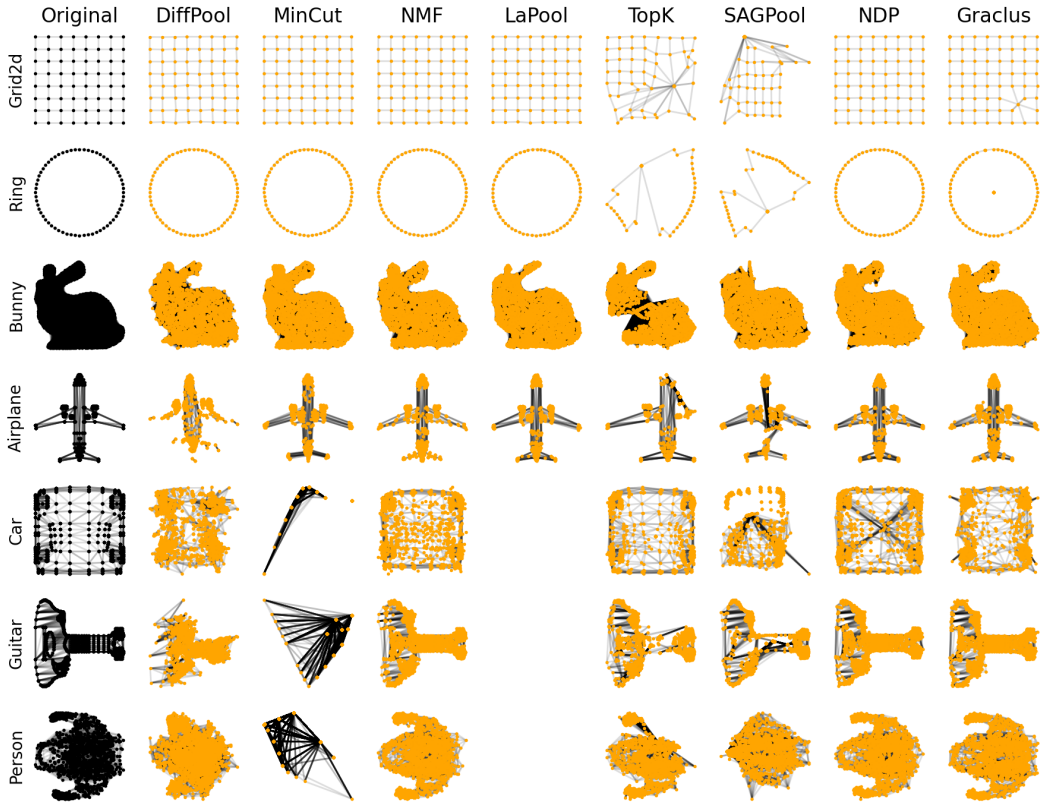


Figure 7: Node attributes (point locations) reconstructed with different operators in the autoencoder experiment.

Table 8: Density and median weight of the edges of the coarsened graphs in the spectral similarity experiment.

		Original	DiffPool	MinCut	NMF	LaPool	TopK	SAGPool	NDP	Graclus
Grid2d	**Density**	0.055	0.969	0.969	0.463	0.917	0.084	0.092	0.189	0.103
	Median	1.000	0.216	0.024	0.018	1.445	1.000	1.000	0.500	0.154
Ring	**Density**	0.031	0.969	0.969	0.125	0.900	0.055	0.061	0.062	0.045
	Median	1.000	0.124	0.032	0.039	1.171	1.000	1.000	0.500	0.250
Bunny	**Density**	0.021	0.999	0.999	0.327	0.952	0.038	0.038	0.104	0.029
	Median	0.812	0.068	$7.55{\cdot}10^{-4}$	$9.99{\cdot}10^{-6}$	234.887	0.815	0.816	0.111	0.026
Minnesota	**Density**	$9.47{\cdot}10^{-4}$	0.999	0.999	0.010	0.999	0.002	0.002	0.003	0.002
	Median	1.000	0.004	$7.58{\cdot}10^{-4}$	0.014	0.013	1.000	1.000	0.333	0.204
Sensor	**Density**	0.159	0.969	0.969	0.844	0.875	0.273	0.230	0.529	0.227
	Median	0.742	0.463	$2.42{\cdot}10^{-4}$	0.005	6.147	0.765	0.756	0.201	0.103
Airfoil	**Density**	0.001	1.000	1.000	0.009	0.996	0.003	0.003	0.006	0.002
	Median	0.500	0.003	$4.70{\cdot}10^{-4}$	0.026	0.209	0.500	0.500	0.090	0.118

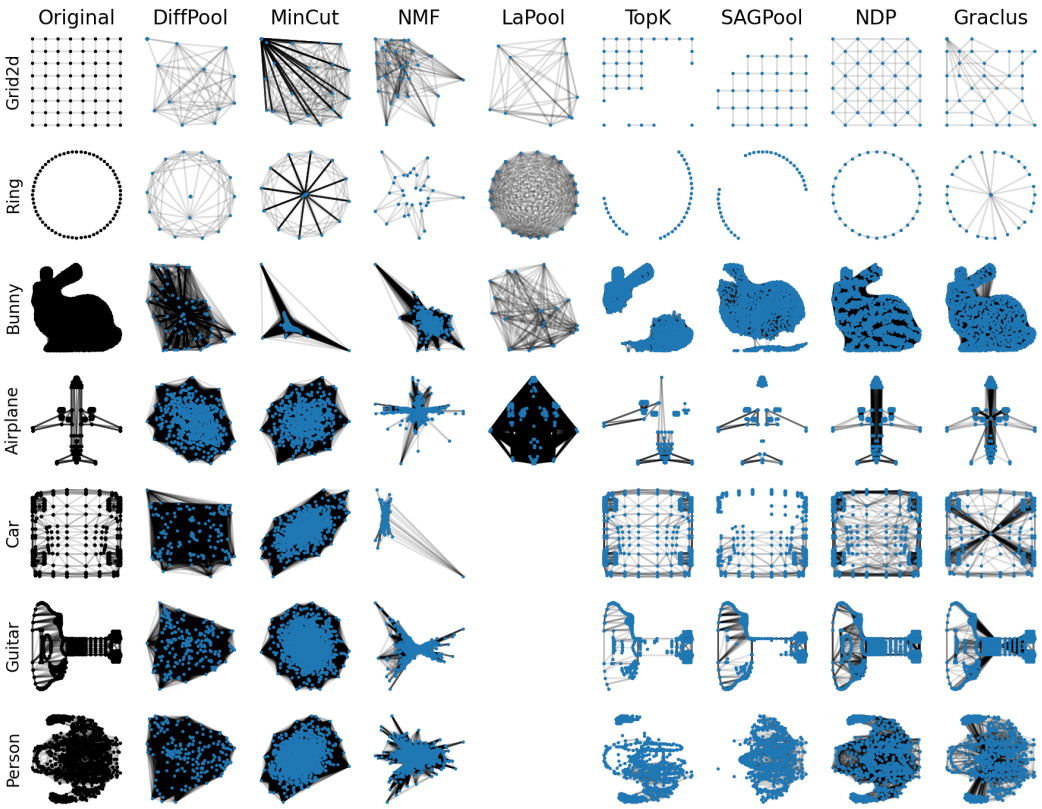


Figure 8: Graphs pooled with different operators in the autoencoder experiment with the modified RED function.

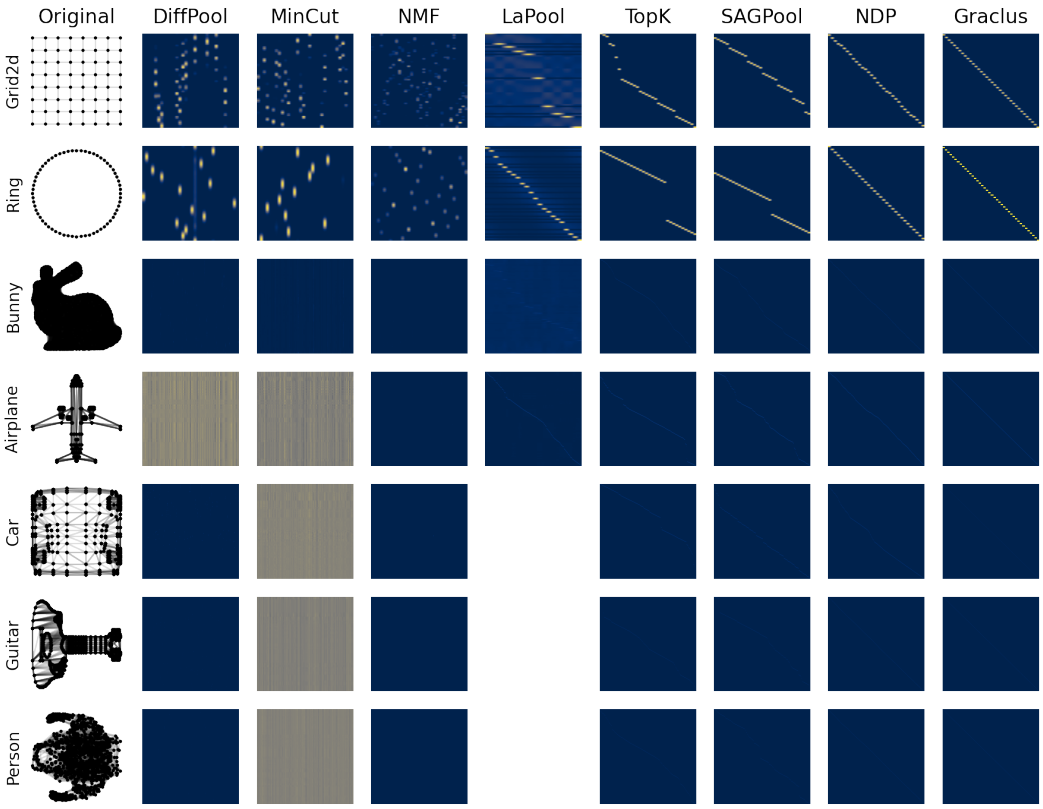
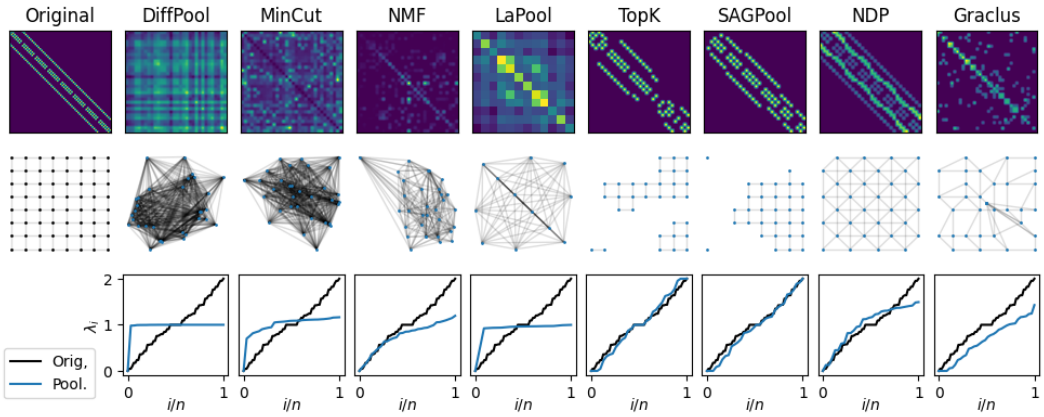
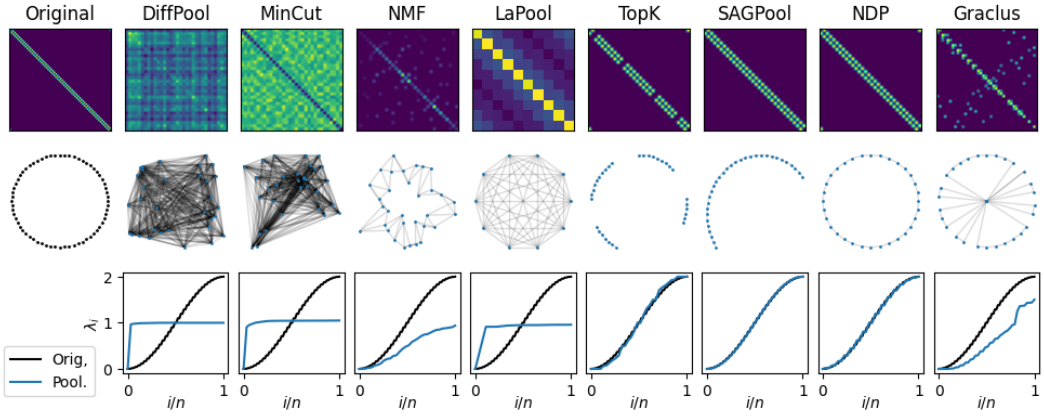


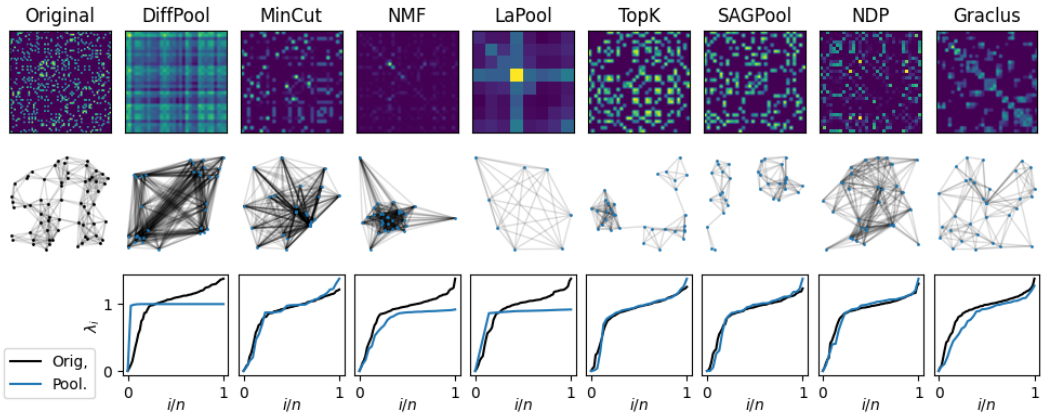
Figure 9: Selection matrices computed with different operators in the autoencoder experiment.



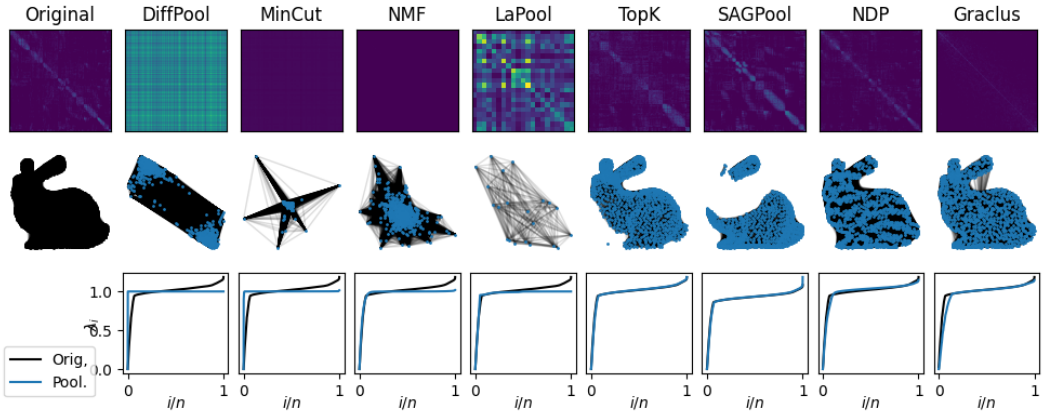
(a) Grid



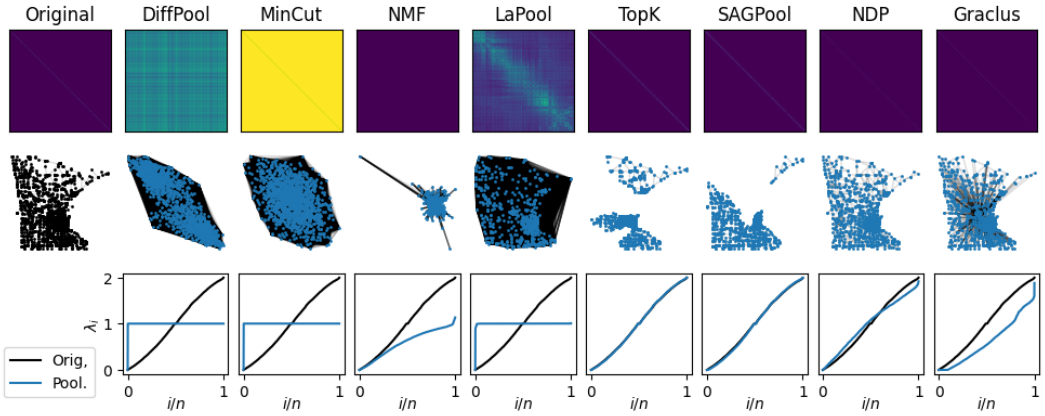
(b) Ring



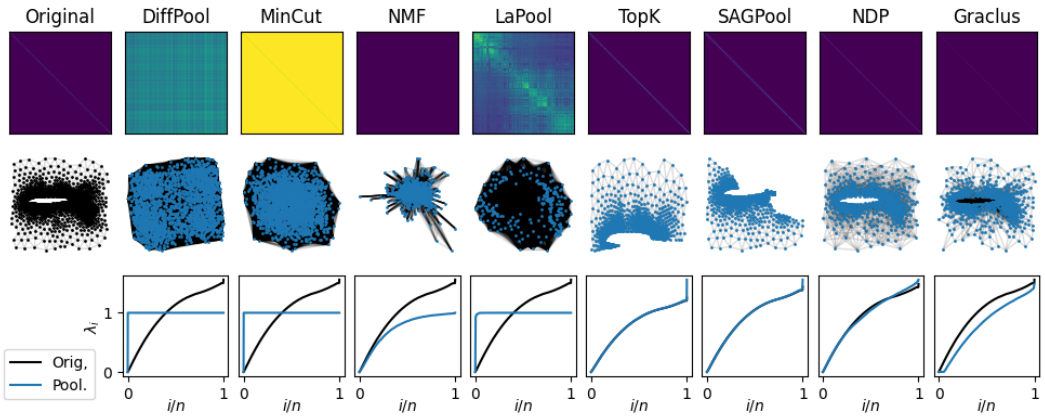
(c) Sensor



(d) Bunny



(e) Minnesota



(f) Airfoil

Figure 10: Results for all graphs when optimizing for spectral similarity. Top: the coarsened adjacency matrices. Middle: the coarsened graphs with modified SEL function. Bottom: the eigenvalues of the normalized Laplacian before (black) and after (blue) pooling. The indices of the eigenvalues are rescaled to fill $[0, 1]$.