Graph Neural Networks with convolutional ARMA filters

Preprint · January 2019

CITATIONS
0

READS
371

4 authors:

Filippo Maria Bianchi
Norce Research
60 PUBLICATIONS **361** CITATIONS
SEE PROFILE

Daniele Grattarola
University of Lugano
7 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Lorenzo Livi
University of Manitoba
78 PUBLICATIONS **751** CITATIONS
SEE PROFILE

Cesare Alippi
Politecnico di Milano
266 PUBLICATIONS **3,659** CITATIONS
SEE PROFILE

Graph Neural Networks with Convolutional ARMA Filters

Filippo Maria Bianchi [1] **Daniele Grattarola** [2] **Cesare Alippi** [2 3] **Lorenzo Livi** [4 5]

Abstract

Recent graph neural networks implement convolutional layers based on polynomial filters operating in the spectral domain. In this paper, we propose a novel graph convolutional layer based on auto-regressive moving average (ARMA) filters that, compared to the polynomial ones, provide a more flexible response thanks to a rich transfer function that accounts for the concept of state. We implement the ARMA filter with a recursive and distributed formulation, obtaining a convolutional layer that is efficient to train, it is localized in the node space and can be applied to graphs with different topologies. In order to learn more abstract and compressed representations in deeper layers of the network, we alternate pooling operations based on node decimation with convolutions on coarsened versions of the original graph. We consider three major graph inference problems: semi-supervised node classification, graph classification, and graph signal classification. Results show that the proposed graph neural network with ARMA filters outperform those based on polynomial filters and sets the new state-of-the-art in several tasks.

1. Introduction

Several deep learning architectures have been proposed to process data represented as graphs. The well-established Convolutional Neural Networks (CNNs) (Krizhevsky et al., 2012) convolve an input tensor with a small trainable kernel of the same rank, applied to fixed-size volumes. Such a bias yields locality and translation invariance in space, which works well for regular grids, but prevents to capture the variability of a graph structure. Therefore, to apply CNNs on graphs, different approaches have been proposed to modify the convolution operations (Atwood & Towsley, 2016; Monti et al., 2017; Fey et al., 2018) or to locally approximate a graph with a regular structure before applying the traditional spatial convolution (Niepert et al., 2016; Zhang et al., 2018).

Graph Neural Networks (GNNs) constitute a class of recently developed tools lying at the intersection between deep learning and methods for structured data, which perform inference on discrete objects (assigned to nodes) by accounting for arbitrary relationships (edges) among them (Battaglia et al., 2018). A GNN combines node features within local neighborhoods on the graph to learn nodes/graph embeddings (Perozzi et al., 2014; Duvenaud et al., 2015; Yang et al., 2016; Hamilton et al., 2017; Bacciu et al., 2018), or to directly perform inference tasks by mapping the node features into categorical labels or real values (Scarselli et al., 2009; Micheli, 2009).

Of particular interest for this work are those GNNs that implement a convolution operation in the spectral domain with a nonlinear trainable filter, which maps the node features in a new space (Bruna et al., 2013; Henaff et al., 2015). To avoid computing the expensive spectral decomposition and projection in the frequency domain, state-of-the-art GNNs approximate graph filters with finite order polynomials (Defferrard et al., 2016; Kipf & Welling, 2016a;b). Polynomial filters have a finite impulse response (FIR) and realize a weighted moving average filtering of graph signals on local node neighbourhoods (Tremblay et al., 2018), thus allowing for fast distributed implementations based on Chebyshev polynomials and Lanczos iterations (Susnjara et al., 2015; Defferrard et al., 2016; Liao et al., 2019). Despite their attractive computational efficiency, FIR filters are sensitive to changes in the graph signal (an instance of the node features) or in the underlying graph structure (Isufi et al., 2016). Moreover, polynomial filters are very smooth and cannot model sharp changes in the frequency response (Tremblay et al., 2018). A more versatile class of filters are the Auto-Regressive Moving Average filters (ARMA) that allow for a more accurate filter design and, in several cases, give exact rather than approximate solutions in modeling the desired response (Narang et al., 2013)

[1]Machine Learning Group, UiT the Arctic University of Tromsø, Norway [2]Faculty of Informatics, Università della Svizzera Italiana, Switzerland [3]Dept. of Electronics, Information, and Bioengineering, Politecnico di Milano, Italy [4]Dept. of Computer Science and Mathematics, University of Manitoba, Canada [5]Dept. of Computer Science, University of Exeter, United Kingdom. Correspondence to: Filippo Maria Bianchi <filippo.m.bianchi@uit.no>.

Contribution In this paper, we address the limitations of existing graph convolutional layers in modeling a desired filter response and propose a GNN based on a novel ARMA layer. The ARMA layer implements a non-linear and trainable ARMA graph filter that generalizes the existing graph convolutional layers based on polynomial filters and provides the GNN with enhanced modeling capability, thanks to a flexible design of the filter transfer function. Contrarily to polynomial filters, ARMA filters are not localized in the node space, making their implementation inefficient within a GNN. To address such a scalability issue, the proposed ARMA layer relies on a recursive formulation, which leads to a fast and distributed implementation that exploits efficient sparse operations on tensors. The resulting filters are not learned in the Fourier space induced by a given Laplacian, but are local in the node space and independent from the underlying graph structure. This allows our GNN to process graphs with different topologies.

We use a node pooling procedure based on node decimation, which builds on the multi-resolution framework adopted in graph signal processing (Shuman et al., 2016). This allows us to build deep architectures that yield more abstract representations at different network depths. Given an input graph, node decimation drops approximately half of the nodes and a coarsened version of the graph on the remaining ones is obtained through graph reduction. Pooling of different strides is implemented in the GNN by means of multiplications with pre-computed matrices.

To assess the performance of our GNN, we apply it to semi-supervised node classification, graph signal classification, and graph classification. Results show that the proposed GNN with ARMA filters outperforms GNNs based on polynomial filters, setting the new state-of-the-art in several tasks.

2. Spectral filtering in GNNs

We assume a graph with M nodes to be characterized by a symmetric adjacency matrix $\mathbf{A} \in \mathbb{R}^{M\times M}$ and we refer to *graph signal* $\mathbf{X} \in \mathbb{R}^{M\times F}$ as the instance of all features (vectors in $\mathbb{R}^F$) associated with the graph nodes. Let $\mathbf{L} = \mathbf{I}_M - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$ be the symmetrically normalized Laplacian ($\mathbf{D}$ is the degree matrix), with spectral decomposition $\mathbf{L} = \sum_{m=1}^{M} \lambda_m \mathbf{u}_m \mathbf{u}_m^T$. A graph filter is a linear operator that modifies the components of $\mathbf{X}$ on the eigenvectors basis of $\mathbf{L}$, according to a transfer function h acting on each eigenvalue λ. The filtered graph signal reads

$$\begin{aligned}\bar{\mathbf{X}} &= \sum_{m=1}^{M} h(\lambda_m)\mathbf{u}_m\mathbf{u}_m^T\mathbf{x}_m, \\ &= \mathbf{U}\,\mathrm{diag}[h(\lambda_1),\dots,h(\lambda_M)]\,\mathbf{U}^T\mathbf{X}\end{aligned} \tag{1}$$

This formulation inspired the seminal work of Bruna et al. (2013) that implemented spectral graph convolutions in a neural network. Their GNN learns end-to-end the parameters of each filter implemented as $h = \mathbf{B}\mathbf{c}$, where $\mathbf{B} \in \mathbb{R}^{M\times K}$ is a cubic B-spline basis and $\mathbf{c} \in \mathbb{R}^K$ is a vector of control parameters. Those filters are not localized, since the full projection of the eigenvectors yields paths of infinite length and the filter accounts for interactions of each node with the whole graph, rather than those limited to the node neighborhood. Since this contrasts with the local design of classic convolutional filters, Henaff et al. (2015) introduced a parametrization of the spectral filters with smooth coefficients to achieve spatial localization. However, the main issue with such spectral filtering (1) is the computational complexity: not only the eigendecomposition of $\mathbf{L}$ is expensive, but a double product with $\mathbf{U}$ must be computed whenever the filter is applied. Notably, $\mathbf{U}$ in (1) is full even when $\mathbf{L}$ is sparse. Finally, the same filter cannot be applied to graphs with different structures since it depends on a specific Laplacian spectrum.

2.1. Chebyshev polynomial filters

The desired transfer function $h(\lambda)$ can be approximated by a polynomial of order K,

$$h_{\text{POLY}}(\lambda) = \sum_{k=0}^{K} w_k \lambda^k, \tag{2}$$

which performs a weighted moving average of the graph signal (Tremblay et al., 2018). Polynomial filters are localized in space, since the output at each node in the filtered signal is a linear combination of the nodes in its K-hop neighbourhood. A localized filter overcomes an important limitation of spectral formulations relying on a fixed Laplacian spectrum, making it suitable also for inference tasks on graphs with different structures (Zhang et al., 2018).

Compared to conventional polynomials, Chebyshev polynomials attenuate unwanted oscillations around the cut-off frequencies (Shuman et al., 2011). Chebyshev polynomials are exploited to implement fast localized filters in a GNN, avoiding to eigen-decompose the Laplacian by approximating the filter convolution with Chebyshev expansion $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ (Defferrard et al., 2016). It follows that the convolutional layers perform the filtering operation

$$\bar{\mathbf{X}} = \sigma\left(\sum_{k=0}^{K-1} T_k(\tilde{\mathbf{L}})\mathbf{X}\mathbf{W}_k\right), \tag{3}$$

where $\tilde{\mathbf{L}} = 2\mathbf{L}/\lambda_{\text{max}} - \mathbf{I}_M$, σ is a non-linear activation (e.g., ReLU), and $\mathbf{W}_k \in \mathbb{R}^{F_{\text{in}}\times F_{\text{out}}}$ are the k trainable weight matrices that map the node's features from an input space F_{in} to a new space F_{out}.

2.2. First-order polynomial filters

A first-order polynomial filter is adopted by Kipf & Welling (2016a) to solve the task of semi-supervised node classification. They propose a GNN called Graph Convolutional Network (GCN), where the convolutional layer is a simplified version of Chebyshev filters

$$\bar{\mathbf{X}} = \sigma\left(\hat{\mathbf{A}}\mathbf{X}\mathbf{W}\right). \tag{4}$$

Their formulation is obtained by (3) considering only $K = 1$ and setting $\mathbf{W} = \mathbf{W}_0 = -\mathbf{W}_1$. Additionally, $\tilde{\mathbf{L}}$ is replaced by $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$, with $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_M$. In respect to $\tilde{\mathbf{L}}$, $\hat{\mathbf{A}}$ contains self-loops that compensate for the removal of the term of order 0 in the polynomial filter, ensuring that a node is part of its 1st order neighbourhood, and that its features are preserved after the convolution. The convolution with higher-order neighbourhoods can be obtained by stacking multiple layers. However, since each layer (4) performs a Laplacian smoothing, after few convolutions the node features becomes too smoothed over the graph (Li et al., 2018)

3. The ARMA graph convolutional layer

The polynomial filters discussed in the previous section are sensitive to changes in the graph signal or in the underlying graph structure, and their smoothness prevents to model filter responses with sharp changes. Moreover, they have poor interpolation and extrapolation capability around the known graph frequencies (Isufi et al., 2016). On the other hand, an ARMA filter approximates better the optimal h thanks to a rational design that allows to model a larger variety of filter shapes (Tremblay et al., 2018). The filter response of an ARMA(P, Q) reads

$$h_{\text{ARMA}}(\lambda) = \frac{\sum_{q=0}^{Q} b_q \lambda^q}{1 + \sum_{p=1}^{P} a_p \lambda^p}, \tag{5}$$

which in the node domain translates to the filtering relation

$$\bar{\mathbf{X}} = \frac{\left(\sum_{q=0}^{Q} b_q \mathbf{L}^q\right)\mathbf{X}}{1 + \sum_{p=0}^{P} a_p \mathbf{L}^p}. \tag{6}$$

The Laplacian appearing in the denominator implies a matrix inversion and multiplication between dense matrices, which is inefficient to implement in a GNN. A possibility is to approximate the inverse with a fixed number of Jacobi iterations, by using a special formulation to avoid numerical instability (Levie et al., 2019b). A more straightforward approach is to completely avoid the inverse computation and consider a distributed formulation (Loukas et al., 2015), which approximates the effect of an ARMA(1,0) filter with a first-order recursion

$$\bar{\mathbf{X}}^{(t+1)} = a\mathbf{M}\bar{\mathbf{X}}^{(t)} + b\mathbf{X}. \tag{7}$$

The eigenvalues of $\mathbf{M} = (\lambda_{\text{max}} - \lambda_{\text{min}})/2\mathbf{I} - \mathbf{L}$ are related to those of the Laplacian $\mathbf{L}$ as follows: $\mu_n = (\lambda_{\text{max}} - \lambda_{\text{min}})/2\lambda_n$. The frequency response of the approximated ARMA(1,0) filter is

$$h_{\text{ARMA}}(\mu) = \frac{r}{\mu - p} \quad \text{with} \quad r = -\frac{b}{a} \quad \text{and} \quad p = \frac{1}{a}. \tag{8}$$

The effect of an ARMA(K,K) filter can be obtained by summing the outputs of K ARMA(1,0) filters

$$\bar{\mathbf{X}} = \sum_{k=1}^{K} \bar{\mathbf{X}}_k = \sum_{k=1}^{K}\sum_{n=1}^{M} \frac{r_k}{\mu_m + p_k} \mathbf{u}_m \mathbf{u}_m^T \mathbf{x}_m. \tag{9}$$

3.1. Recursive and distributed implementation of the ARMA layer

Here we propose a recursive implementation of the ARMA(K,K) filter based on neural networks; see Fig. 1. Equation (7) must be applied many times before converging to a steady state. Instead, to obtain a more efficient implementation, we apply the recursive update only a few times and compensate by adding a non-linearity and trainable parameters.

We implement the recursive update in (7) with a *Graph Convolutional Skip* (GCS) layer, defined as

$$\bar{\mathbf{X}}^{(t+1)} = \sigma\left(\tilde{\mathbf{L}}\bar{\mathbf{X}}^{(t)}\mathbf{W}^{(t)} + \mathbf{X}\mathbf{V}^{(t)}\right), \tag{10}$$

where $\mathbf{W}^{(t)} \in \mathbb{R}^{F_{\text{out}}^t \times F_{\text{out}}^{t+1}}$ and $\mathbf{V}^{(t)} \in \mathbb{R}^{F_{\text{in}} \times F_{\text{out}}^{t+1}}$ are trainable parameters; we set $\bar{\mathbf{X}}^{(0)} = \mathbf{X}$. The modified Laplacian matrix $\tilde{\mathbf{L}} = \mathbf{I} - \mathbf{L}$ is derived by setting $\lambda_{\text{min}} = 0$ and $\lambda_{\text{max}} = 2$ in $\mathbf{M}$. This is a reasonable simplification, since the spectrum of $\mathbf{L}$ lies in $[0, 2]$ and the trainable parameters in $\mathbf{W}^{(t)}$ can adjust the small offset introduced. Each GCS layer extracts local substructure information by aggregating node information in local neighbourhoods and, through the skip connection, by combining them with the original node features. If $\mathbf{L}$ and/or $\mathbf{X}$ are represented by sparse tensors, the GCS can exploit efficient sparse operations.

We build K parallel stacks, each one with T GCS layers, and define the output of the ARMA convolutional layer as

$$\bar{\mathbf{X}} = \text{avgpool}\left(\sum_{k=1}^{K} \bar{\mathbf{X}}_k^{(T)}\right), \tag{11}$$

where $\bar{\mathbf{X}}_k^{(T)}$ is the last output of the k-th stack. We apply dropout to the skip connection of each GCS layer not only for regularization, but also to encourage diversity in the filters learned in each one of the K parallel stacks. To provide a further regularization and reduce the number of parameters in the ARMA layer, the GCS layers in each stack may share the same parameters, except for $\mathbf{W}_k^{(1)} \in \mathbb{R}^{F_{\text{in}} \times F_{\text{out}}}$

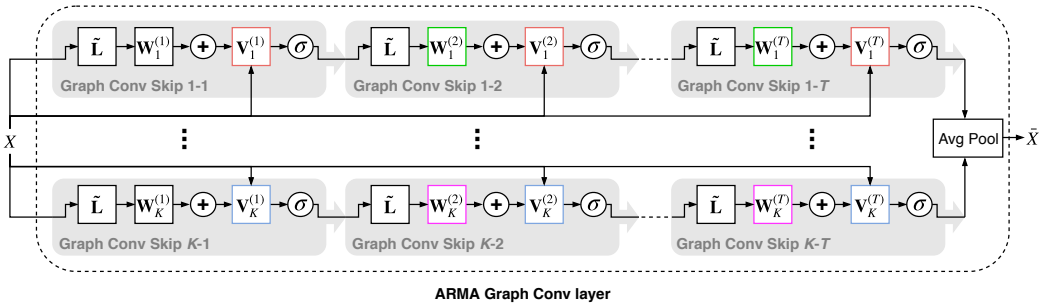

Figure 1. The ARMA convolutional layer. Same colour indicates shared weights.

that performs a different mapping in the first layer of the stack. Namely, $\mathbf{W}_k^{(i)} = \mathbf{W}_k^{(i+1)} = \mathbf{W}_k \in \mathbb{R}^{F_{\text{out}} \times F_{\text{out}}}, \forall i > 1$ and $\mathbf{V}_k^{(i)} = \mathbf{V}_k^{(i+1)} = \mathbf{V}_k \in \mathbb{R}^{F_{\text{in}} \times F_{\text{out}}}, \forall i$. Since each stack of GCS layers is executed independently from the others, it is possible to implement the ARMA layer in a distributed fashion using multiple GPUs.

3.2. Properties and relationship with other approaches

Contrarily to filters defined in the spectral domain (Bruna et al., 2013), ARMA filters do not explicitly depend on eigenvectors and eigenvalues, making them robust to perturbations in the underlying graph. Hence, as formally proven in (Levie et al., 2019a), the proposed ARMA filters based on rational functions are stable and transferable, i.e., they can generalize to graph signals not seen during training and graphs with different topologies. Contrarily to polynomial filters, in the ARMA layer $\mathbf{L}$ is not exponentiated and remains sparse; this implies faster computations since $\mathbf{L}$ becomes quickly dense ($\mathbf{L}^P$ describes a fully connected graph if P is the longest shortest path in $\mathbf{L}$). The ARMA layer can naturally deal with time-varying graph signals (Holme, 2015; Grattarola et al., 2018) by replacing the constant term $\mathbf{X}$ in (10) with a time-dependent input $\mathbf{X}^{(t)}$. The GCS layer has a similar formulation to the graph convolutional layer in (4); however, thanks to the skip connection, it is possible to stack multiple layers without risking to over-smooth the node features (Li et al., 2018). The formulation of the ARMA layer with shared weights is similar to recurrent neural networks with residual connections (Wu et al., 2016). Finally, similarly to GNNs operating in the node domain (Scarselli et al., 2009; Gallicchio & Micheli, 2010), each GCS layer computes the filtered signal $\bar{\mathbf{x}}_i^{(t+1)}$ at vertex i as a combination of signals $\mathbf{x}_j^{(t)}$ in its 1-hop neighborhood, $j \in \mathcal{N}(i)$; such a commutative aggregation solves the problem of undefined vertex ordering and varying neighborhood sizes.

4. Node Pooling

Node pooling associates a single label to the node features and is particularly important in tasks such as graph (signal) classification. However, contrarily to other neural networks, GNNs also require to coarsen the original graph to perform further convolutions on graph signals as the node dimensionality is reduced through the network layers.

A recent approach (Ying et al., 2018) proposes to learn differentiable soft assignments to cluster the nodes at each layer. The original adjacency matrix acts as a prior when learning the soft assignment and sparsity is enforced with an entropy-based regularization. However, the application of this method to medium and large graphs is not feasible, as it introduces a number of additional trainable parameters quadratic in the number of nodes. The other approach followed in most GNNs consists of pre-computing coarsened versions of the graph using hierarchical clustering (Bruna et al., 2013; Defferrard et al., 2016; Monti et al., 2017; Fey et al., 2018). At each level l, two vertices $x_i^{(l)}$ and $x_j^{(l)}$ are clustered together in a new vertex $x_z^{(l+1)}$. Then, a standard pooling operation is applied to halve the size of the graph signal. To make the pooling output consistent with the cluster assignment, the graph signal is rearranged so that elements i and j end up in consecutive positions. This approach has several drawbacks. First, the connectivity of the original graph is not preserved in the coarsened graphs and the spectrum of their associated Laplacians is usually not contained in the spectrum of the original Laplacian. Second, the procedure to rearrange vertices is cumbersome to implement; moreover, it requires to add fake vertices so that the number of nodes can be halved each time, hence injecting noisy information in the graph signal. Finally, clustering results depend on the initial nodes order of the nodes, which hampers stability and reproducibility.

In this paper, we use a pooling procedure that builds on the multi-resolution framework adopted in graph signal processing (Shuman et al., 2016), which addresses the drawbacks of the aforementioned methods. A similar, yet preliminary approach was recently discussed by Simonovsky & Komodakis (2017). Here, we provide a more detailed formulation framed within the GNN framework of the pooling procedure based on node decimation, and of the graph reduction to generate a new coarsened graph, necessary to apply graph convolutions in the next GNN layer. In the experiments, we provide a systematic comparison with

respect to pooling methods based on graph clustering.

4.1. Node decimation pooling and graph reduction

Pooling with node decimation. A simple way to decimate nodes $\mathcal{V}$ of an arbitrary graph consists of partitioning them in two sets based on Fiedler vector $\mathbf{u}_{\max}$ of the Laplacian, and then drop one of the two sets of nodes (Shuman et al., 2016). In particular, the pooling operation keeps only the nodes in $\mathcal{V}^+$, defined as

$$\mathcal{V}^+ = \{n \in \mathcal{V} : \mathbf{u}_{\max}(n) \geq 0\}. \tag{12}$$

We note that is equivalent to keep each time the nodes in $\mathcal{V}^-$, i.e., those associated with a negative value in $\mathbf{u}_{\max}$. Despite its simplicity, this procedure offers important advantages: i) approximately half of the nodes are removed each time, i.e., $|\mathcal{V}^+| \approx |\mathcal{V}^-|$; ii) the nodes in $\mathcal{V}^+$ and $\mathcal{V}^-$ are connected by edges with small weights; iii) the Fiedler vector can be quickly computed with the power method. Furthermore, compared to the pooling based on graph clustering, this approach avoids to introduce fake nodes and to reorder nodes according to their cluster indices.

The pooling operation is implemented by multiplying a graph signal $\mathbf{X}$ with a *decimation matrix* $\mathbf{S}$, which is obtained by keeping in the identity matrix $\mathbf{I}_M$ only the rows corresponding to the vertices in $\mathcal{V}^+$,

$$\mathbf{X}_{\text{pool}} = \mathbf{S}\mathbf{X} = [\mathbf{I}_M]_{\mathcal{V}^+,\mathcal{V}}\,\mathbf{X}. \tag{13}$$

Graph reduction. A simple approach to reduce the original Laplacian to a new Laplacian $\mathbf{L}_{\text{new}}$ defined on the subset $\mathcal{V}^+$ consists in computing

$$\mathbf{L}_{\text{new}} = ([\mathbf{L}]^2)_{\mathcal{V}^+,\mathcal{V}^+}, \tag{14}$$

which are the selected rows and columns of the 2-hop Laplacian (Narang & Ortega, 2010). Since the decimation operation ideally removes the first-closest neighbour of nodes $\mathcal{V}^+$ (i.e., the nodes in $\mathcal{V}^-$), it is intuitive that before being dropped the nodes should propagate their information in the first-order neighbourhood. While this graph reduction is very fast to compute, it does not always preserve connectivity, introduces self-loops, and the spectra of $\mathbf{L}$ and $\mathbf{L}_{\text{new}}$ might not be interlaced (i.e., the spectrum of $\mathbf{L}_{\text{new}}$ is not always contained in the spectrum of $\mathbf{L}$).

The Kron reduction (Shuman et al., 2016) is a more advanced technique that defines the reduced Laplacian as

$$\mathbf{L}_{\text{new}} = \mathbf{L}_{\mathcal{V}^+,\mathcal{V}^+} - \mathbf{L}_{\mathcal{V}^+,\mathcal{V}^-}\mathbf{L}^{-1}_{\mathcal{V}^-,\mathcal{V}^-}\mathbf{L}_{\mathcal{V}^-,\mathcal{V}^+} \tag{15}$$

The resulting $\mathbf{L}_{\text{new}}$ is a well-defined Laplacian where two nodes are connected only if there is a path between them in the original $\mathbf{L}$. Furthermore, $\mathbf{L}_{\text{new}}$ does not introduce self-loops and guarantees spectral interlacing and resistance distance preservation (Shuman et al., 2016). The main drawback compared to (14) is the computation of the inverse, which can give memory issues in very large graphs. Due to the connectivity preservation property, $\mathbf{L}_{\text{new}}$ becomes denser after each Kron reduction. Since graph convolutions are implemented by sparse operations, this implies that deeper layers will require more computation. A solution is to apply after each reduction spectral sparsification (Batson et al., 2013) on $\mathbf{L}_{\text{new}}$. However, we experienced numerical instability and poor convergence when applying the sparsification algorithm. Therefore, we opted for dropping connections with weights below a small threshold (1E-4), which keeps the desired level of sparsity in $\mathbf{L}_{\text{new}}$ without altering too much its spectrum.

Pooling with larger stride.

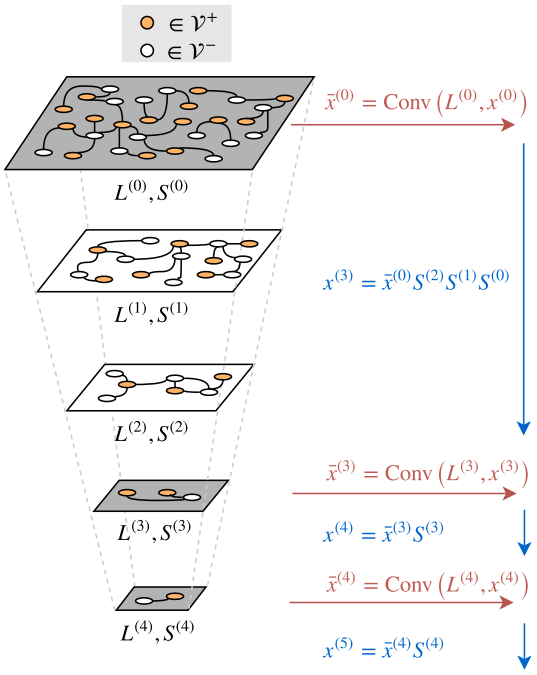

Figure 2. How to perform convolutions with selected Laplacians in the pyramid by means of higher order pooling.

It is possible to perform convolutions only with some Laplacians in the pyramid and apply pooling with larger stride to transit from level i to level $i + k$, with $k > 1$. The application of a single decimation matrix $\mathbf{S}^{(i)}$ corresponds to a classic pooling with stride 2, as it approximately halves the number of nodes. A pooling with stride 2^k is obtained by applying k decimation matrices in cascade. Fig. 2 depicts an example of a pooling with approximately stride 8, which allows to skip 2 levels in the pyramid and to apply directly a convolution with the Laplacian $\mathbf{L}^{(3)}$ after the first convolution with $\mathbf{L}^{(0)}$.

5. Experiments

We consider three classification tasks on graph data: node classification, graph signal classification, and graph classification. Since we process only graphs of medium and small size, in all experiments we use Kron reduction (15). However, we advise the reduction in (14) when dealing with very large graphs to avoid memory issues.

5.1. Semi-supervised node classification

The input for this task is a single graph described by an adjacency matrix $\mathbf{A} \in \mathbb{R}^{M \times M}$, a graph signal $\mathbf{X} \in \mathbb{R}^{M \times F_{\text{in}}}$ and the labels $\mathbf{y}_l \in \mathbb{R}^{M_l}$ of a subset of nodes $M_l \subset M$. The target outputs are the labels $\mathbf{y}_u \in \mathbb{R}^{M_u}$ of the unlabelled nodes. For this task, pooling is not required since the output is computed in the input node space by mapping nodes features into labels through graph convolutions.

We follow the same experimental setup of (Kipf & Welling, 2016a) applied to three citation network datasets: Citeseer, Cora and Pubmed. Each dataset is a graph, whose nodes $\mathbf{x} \in \mathbb{R}^{F_{\text{in}}}$ are documents represented by sparse bag-of-words feature vectors. The binary undirected edges in $\mathbf{A}$ indicate citation links between documents. For training, 20 labels per document class are used ($\mathbf{y}_l$) and the performance is evaluated as classification accuracy on $\mathbf{y}_u$.

As in (Kipf & Welling, 2016a), we use a 2-layers GNN with 16 hidden units and we report in Tab. 2 the mean classification accuracy obtained for different graph convolutional layers: the ones based on Chebyshev polynomials (Cheby), their first order approximation (GCN), and the proposed ARMA layers. Tab. 1 reports the hyperparameters configuration found with cross-validation: L_2 regularization weight, dropout probability (p_{drop}), number of stacks (K) and depth (T) in the ARMA filter, and usage of shared weights in the GCS layer. As additional baselines, we include the results from the literature obtained by Label Propagation (LP) (Zhou et al., 2004), Deepwalk (DW) (Perozzi et al., 2014), Planetoid (PL) (Yang et al., 2016), Graph Attention Networks (GAT) (Velickovic et al., 2017), and LanczosNets (LNet) (Liao et al., 2019).

Table 1. Hyperparameters setting for node classification

Dataset	L_2 reg.	p_{drop}	$[K, T]$	shared $\mathbf{W}$
Cora	5e-4	0.25	[3,2]	yes
Citeseer	5e-4	0.75	[3,1]	yes
Pubmed	5e-4	0.0	[1,4]	no

Node classification is a semi-supervised task that requires a strong regularization and a simple model to avoid overfitting on the few labels available. This is the key of the success of the GCN model compared to the more complex Chebyshev filters. However, despite the more powerful modelling capability, thanks to its flexible formulation the proposed ARMA layer can implement the right degree of complexity for each task and outperforms other approaches. Notably, our method surpasses even GAT, which exploits a sophisticated attention mechanism to learn how to weight each link when applying the graph convolution.

Table 2. Mean node classification accuracy

Method	**Cora**	**Citeseer**	**Pubmed**
LP	68.0	45.3	63.0
DW	67.2	43.2	65.3
PL	75.7	64.7	77.2
GAT	83.0	72.5	79.0
LNet	80.4	68.7	78.3
Cheby	81.2	69.8	74.4
GCN	81.5	70.3	79.0
ARMA (ours)	**84.7**	**73.8**	**81.4**

5.2. Graph signal classification

In this task, N different graph signals $\mathbf{X} \in \mathbb{R}^{M \times F_{\text{in}}}$, defined on the same adjacency matrix $\mathbf{A} \in \mathbb{R}^{M \times M}$, must be classified with labels $\mathbf{y}_1, \dots, \mathbf{y}_N$. Like in traditional CNNs, this task can be solved by a deep architecture composed of L graph convolutional layers, each one followed by a pooling layer. In each layer l, the graph convolution modifies the vertex features by mapping the graph signal $\mathbf{x}^{(l)} \in \mathbb{R}^{M_l \times F_l}$ into $\bar{\mathbf{x}}^{(l)} \in \mathbb{R}^{M_l \times F_{(l+1)}}$, while the pooling operation maps $\bar{\mathbf{x}}^{(l)}$ into a new node space $\mathbf{x}^{(l+1)} \in \mathbb{R}^{M_{l+1} \times F_{l+1}}$. In the last layer, the features of the remaining nodes are aggregated by a global operation, $\mathbf{x} \in \mathbb{R}^{M_L \times F_L} \to \mathbf{x} \in \mathbb{R}^{F_L}$, and a Softmax layer is applied to compute the labels. We perform experiments following the same setting of (Defferrard et al., 2016) on the MNIST and 20news datasets and, unless specified otherwise, we use the same hyperparameters.

MNIST. To emulate a classic CNNs operating on a regular 2D grid, an 8-NN graph is defined on the 784 pixel positions of the MNIST images. The elements in $\mathbf{A}$ are

$$a_{ij} = \exp\left(-\frac{\|p_i - p_j\|^2}{\sigma^2}\right), \tag{16}$$

where p_i and p_j are the 2D coordinates of pixel i and j. Each graph signal is a vectorized image $\mathbf{x} \in \mathbb{R}^{784 \times 1}$. As network architecture, we use GC16-P4-GC32-P4-FC512, where GC16 and GC32 indicate a graph convolutional layer with 16 and 32 hidden units respectively, P4 a pooling operation with stride 4, and FC512 a fully connected layer with 512 units. Compared to (Defferrard et al., 2016), we use less hidden units to diversify more the results for different filters and pooling methods. The ARMA filters are configured with $K = 5$, $T = 10$, and no shared weights. As discussed in Sect. 4.1, when using decimation pooling

a stride 4 is approximated by two decimation matrices in cascade ($\mathbf{S}^{(1)}\mathbf{S}^{(0)}$ and $\mathbf{S}^{(3)}\mathbf{S}^{(2)}$ in this case).

Table 3. Graph signal classification results on MNIST.

GC layer	Pooling	
	clust	decim(k)
GCN	97.57	95.91
Cheby	98.17	97.64
ARMA (ours)	**98.54**	**98.11**

The results, reported in Tab. 3, show that a GNN with ARMA filters achieves the best results. On the other hand, the GCN layers yield the worst performance, suggesting that for more complex graph signal classification tasks their simple formulation is not sufficient. Also, the GNN performs better when using the hierarchical clustering pooling (Defferrard et al., 2016), rather than the node decimation pooling. This is expected since the artificial 8-NN graph generated for this task, contrarily to most real-world graphs, is extremely regular and the node pairs are easily matched by the clustering procedure.

20news. The dataset consists of 18,846 documents from 20 classes. Each graph signal is a document that is represented by a bag-of-words of the 10,000 most frequent words in the corpus. Each word is, in turn, represented by a word2vec embedding. The underlying graph of 10,000 nodes is defined by a 16-NN adjacency matrix built with (16), where p_i, p_j are the embeddings of words i and j.

Table 4. Graph signal classification results on 20news.

Method	Accuracy
Linear SVM	65.90
Multinomial Naive Bayes	68.51
Softmax	66.28
GCN	65.57
Cheby	68.26
ARMA (ours)	**70.12**

Tab. 4 reports the average classification accuracy obtained by a GNN with a single conv layer, followed by global average pooling and Softmax. We report all the results from (Defferrard et al., 2016), and we compare them with those obtained using a GCN and the proposed ARMA layer. The ARMA layer has 16 hidden units and is configured with K=1, T=1, 1E-3 as L_2 regulariazion, and 0.75 dropout. For the GCN layer we used 32 hidden units, which is the same number of units for the Chebyshev layer in (Defferrard et al., 2016). The GNN with GCN layer performs worse than any method. On the other hand, the proposed ARMA GNN outperforms Chebyshev GNN and also every other model. Since we use only one GCS layer (K=1), the main difference between the GCN and our layer is the presence of the skip connection with high dropout, which turns out to be extremely important for the inference task.

5.3. Graph classification

In this task, the ith datum is a graph represented by a pair $\{\mathbf{A}_i, \mathbf{X}_i\}, i = 1, \dots N$, where $\mathbf{A}_i \in \mathbb{R}^{M_i \times M_i}$ is an adjacency matrix with M_i nodes and the graph signal $\mathbf{X}_i \in \mathbb{R}^{M_i \times F}$ describes the node features. Each sample must be classified with a label $\mathbf{y}_i$. To train the GNN on mini-batches of graphs with a variable number of nodes, we compute the disjoint union of the graphs in each mini-batch, and train the network on the obtained Laplacian and graph signal. In this way, we can apply the convolution and pooling operations seamlessly, performing batched computations on GPU. At the end, an average pooling matrix aggregates the features on the remaining nodes in each graph signal, and a Softmax layer yields the final output for each graph. Fig. 3 depicts an example of the procedure.

To test our model, we consider 4 datasets from the benchmark database for graph kernels[1]: Enzymes, Proteins, D&D, and MUTAG. We used node degree, clustering coefficients, and node labels as additional node features. For each experiment we adopted a fixed architecture, which is GC64-P2-GC64-P2-GC64-P2-AvgPool-Softmax. Such a configuration might not be optimal for all dataset, but the main focus of this experiment is to compare on a common ground the different graph filters and the pooling procedures based on node decimation and hierarchical graph clustering. Tab. 5 reports the optimal configurations of ARMA and Cheby filters found with cross-validation on each dataset. To evaluate model performance we perform a 10-fold train/test split, using 10% of the training set in each fold as validation set, and in Tab. 6 we report the accuracy averaged over 10 folds. For comparison, we also add in Tab. 6 the results obtained by state-of-the-art graph kernels and other neural networks for graph classification: the Weisfeiler-Lehman kernel (WL) (Shervashidze et al., 2011); the Edge-Conditioned Convolution network (ECC) (Simonovsky & Komodakis, 2017); PATCHY-SAN (Niepert et al., 2016); GRAPHSAGE (Hamilton et al., 2017); the Diffusion-CNN (DCNN) (Atwood & Towsley, 2016); the network with differential pooling (DIFFPOOL) (Ying et al., 2018); the Deep Graph Convolutional Neural Network (DGCNN) (Zhang et al., 2018).

GCN performs better than Cheby only on the Protein dataset, while the proposed ARMA layer always achieves the best performance showing, once again, a superior modeling capability compared to those layers based on polynomial filters. The adopted GNN architecture is particu-

[1] https://ls11-www.cs.tu-dortmund.de/staff/morris/graphkerneldatasets

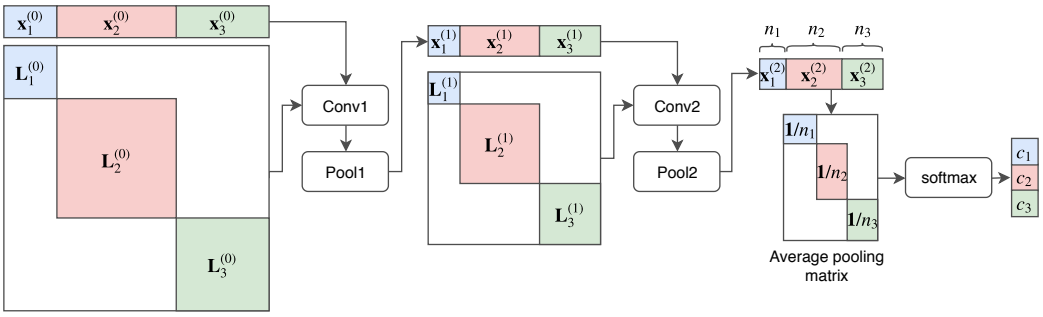

Figure 3. Implementation of graph classification with mini batches.

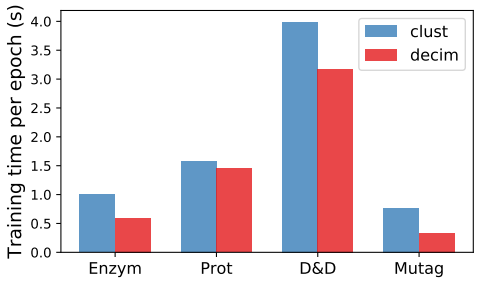

Figure 4. Mean training time per epoch when using ARMA filters and pooling based on hierarchical clustering or decimation.

Table 5. Hyperparameters setting for graph classification

Dataset	L_2 reg.	p_{drop}	**Cheby** K	**ARMA** $[K, T]$	**ARMA** shared $\mathbf{W}$
Enzymes	5e-4	0.5	3	[1,4]	yes
Proteins	5e-4	0.5	10	[3,2]	no
D&D	5e-4	0.0	5	[3,4]	yes
MUTAG	5e-4	0.25	10	[3,4]	no

Table 6. Graph classification results

	Method	**Enzymes**	**Protein**	**D&D**	**MUTAG**
	WL	53.53	72.92	74.02	80.72
	ECC	53.50	72.65	74.10	89.44
	PATCHY-SAN	–	75.00	76.27	92.63
	GRAPHSAGE	54.25	70.48	75.42	–
	DCNN	18.10	61.29	58.09	66.98
	DIFFPOOL	62.53	**76.25**	**80.64**	–
	DGCNN	–	75.54	79,73	85.83
clust	GCN	64.83	72.06	64.60	76.13
clust	Cheby	66.50	69.19	66.81	80.32
clust	**ARMA (ours)**	67.83	71.92	71.22	85.67
decim	GCN	67.33	72.15	70.63	86.20
decim	Cheby	66.50	70.79	68.09	90.39
decim	**ARMA (ours)**	**69.66**	75.12	74.86	**93.25**

larly effective for the Enzymes dataset, as it surpasses the state-of-the-art with every convolutional layer and pooling method. The GNN is configured with ARMA layers and decimation pooling attains top performance also in MUTAG, and competitive results in Protein. Finally, in D&D results are below the state-of-the-art, suggesting that the adopted architecture (GC64-P2-GC64-P2-GC64-P2-AvgPool-Softmax) is not optimal for this task.

Contrarily to the results obtained on the artificial grid for the MNIST graph signal classification, here the decimation pooling always outperforms the clustering pooling. This demonstrates that for irregular graph structures with a variable number of nodes, the node decimation pooling is much more effective. Moreover, Fig. 4 shows that, when using decimation pooling, training GNN is faster. Indeed, in cluster pooling fake nodes must be added whenever the number of nodes is not divisible by 2^L (in our case $L = 3$, since we apply pooling 3 times), which implies larger graphs and slower convolutions.

6. Conclusions

We proposed a recursive formulation of the ARMA graph convolutional layer, which allows for a fast and distributed GNN implementation that exploits efficient sparse tensor operations to perform graph convolutions with the Laplacians. The proposed ARMA layer outperformed existing convolutional layers based on polynomial filters on all classification tasks on graph data taken into account. To build a deep GNN, we used a pooling operation based on node decimation, which achieves superior performance on real-world graphs with irregular topology and faster training time compared to node pooling based on graph clustering.

The current formulation of the ARMA layer only considers nodes information, but can be extended to incorporate edge features to weight the contribution of each neighbor node using, for example, edge-conditioned convolutions (Simonovsky & Komodakis, 2017). Moreover, the results presented in (Velickovic et al., 2017) showed a notable increase in performance when applying multi-head soft attention to the Laplacian in a GNN. Given that the ARMA layer is already structured in a parallel fashion, a similar extension with the attention mechanism could provide comparable benefits, and further improve the performance.

References

Atwood, James and Towsley, Don. Diffusion-convolutional neural networks. In *Advances in Neural Information Processing Systems*, pp. 1993–2001, 2016.

Bacciu, Davide, Errica, Federico, and Micheli, Alessio. Contextual graph markov model: A deep and generative approach to graph processing. In *Proceedings of the 35th international conference on Machine learning*. ACM, 2018.

Batson, Joshua, Spielman, Daniel A, Srivastava, Nikhil, and Teng, Shang-Hua. Spectral sparsification of graphs: theory and algorithms. *Communications of the ACM*, 56 (8):87–94, 2013.

Battaglia, Peter W, Hamrick, Jessica B, Bapst, Victor, Sanchez-Gonzalez, Alvaro, Zambaldi, Vinicius, Malinowski, Mateusz, Tacchetti, Andrea, Raposo, David, Santoro, Adam, Faulkner, Ryan, et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bruna, Joan, Zaremba, Wojciech, Szlam, Arthur, and LeCun, Yann. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Defferrard, Michaël, Bresson, Xavier, and Vandergheynst, Pierre. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pp. 3844–3852, 2016.

Duvenaud, David K, Maclaurin, Dougal, Iparraguirre, Jorge, Bombarell, Rafael, Hirzel, Timothy, Aspuru-Guzik, Alán, and Adams, Ryan P. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in neural information processing systems*, pp. 2224–2232, 2015.

Fey, Matthias, Lenssen, Jan Eric, Weichert, Frank, and Müller, Heinrich. Splinecnn: Fast geometric deep learning with continuous b-spline kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 869–877, 2018.

Gallicchio, Claudio and Micheli, Alessio. Graph echo state networks. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1–8. IEEE, 2010.

Grattarola, Daniele, Zambon, Daniele, Alippi, Cesare, and Livi, Lorenzo. Change detection in graph streams by learning graph embeddings on constant-curvature manifolds. *arXiv preprint arXiv:1805.06299*, 2018.

Hamilton, Will, Ying, Zhitao, and Leskovec, Jure. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017.

Henaff, Mikael, Bruna, Joan, and LeCun, Yann. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163*, 2015.

Holme, Petter. Modern temporal network theory: a colloquium. *The European Physical Journal B*, 88(9):234, 2015.

Isufi, Elvin, Loukas, Andreas, Simonetto, Andrea, and Leus, Geert. Autoregressive moving average graph filtering. *arXiv preprint arXiv:1602.04436*, 2016.

Kipf, Thomas N and Welling, Max. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2016a.

Kipf, Thomas N and Welling, Max. Variational graph auto-encoders. In *NIPS Workshop on Bayesian Deep Learning*, 2016b.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Levie, Ron, Elvin, Isufi, and Gitta, Kutyniok. On the transferability of spectral graph filters. *arXiv preprint*, 2019a.

Levie, Ron, Monti, Federico, Bresson, Xavier, and Bronstein, Michael M. Cayleynets: Graph convolutional neural networks with complex rational spectral filters. *IEEE Transactions on Signal Processing*, 67(1):97–109, Jan 2019b. ISSN 1053-587X. doi: 10.1109/TSP.2018.2879624.

Li, Qimai, Han, Zhichao, and Wu, Xiao-Ming. Deeper insights into graph convolutional networks for semi-supervised learning. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2018.

Liao, Renjie, Zhao, Zhizhen, Urtasun, Raquel, and Zemel, Richard. Lanczosnet: Multi-scale deep graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2019.

Loukas, Andreas, Simonetto, Andrea, and Leus, Geert. Distributed autoregressive moving average graph filters. *IEEE Signal Processing Letters*, 22(11):1931–1935, 2015.

Micheli, Alessio. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

Monti, Federico, Boscaini, Davide, Masci, Jonathan, Rodola, Emanuele, Svoboda, Jan, and Bronstein, Michael M. Geometric deep learning on graphs and

manifolds using mixture model cnns. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, volume 1, pp. 3, 2017.

Narang, Sunil K and Ortega, Antonio. Local two-channel critically sampled filter-banks on graphs. In *Image Processing (ICIP), 2010 17th IEEE International Conference on*, pp. 333–336. IEEE, 2010.

Narang, Sunil K, Gadde, Akshay, and Ortega, Antonio. Signal processing techniques for interpolation in graph structured data. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 5445–5449. IEEE, 2013.

Niepert, Mathias, Ahmed, Mohamed, and Kutzkov, Konstantin. Learning convolutional neural networks for graphs. In *International conference on machine learning*, pp. 2014–2023, 2016.

Perozzi, Bryan, Al-Rfou, Rami, and Skiena, Steven. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710. ACM, 2014.

Scarselli, Franco, Gori, Marco, Tsoi, Ah Chung, Hagenbuchner, Markus, and Monfardini, Gabriele. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Shervashidze, Nino, Schweitzer, Pascal, Leeuwen, Erik Jan van, Mehlhorn, Kurt, and Borgwardt, Karsten M. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.

Shuman, David I, Vandergheynst, Pierre, and Frossard, Pascal. Chebyshev polynomial approximation for distributed signal processing. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pp. 1–8. IEEE, 2011.

Shuman, David I, Faraji, Mohammad Javad, and Vandergheynst, Pierre. A multiscale pyramid transform for graph signals. *IEEE Transactions on Signal Processing*, 64(8):2119–2134, 2016.

Simonovsky, Martin and Komodakis, Nikos. Dynamic edge-conditioned filters in convolutional neural networks on graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

Susnjara, Ana, Perraudin, Nathanael, Kressner, Daniel, and Vandergheynst, Pierre. Accelerated filtering on graphs using lanczos method. *arXiv preprint arXiv:1509.04537*, 2015.

Tremblay, Nicolas, Goncalves, Paulo, and Borgnat, Pierre. Design of graph filters and filterbanks. In *Cooperative and Graph Signal Processing*, pp. 299–324. Elsevier, 2018.

Velickovic, Petar, Cucurull, Guillem, Casanova, Arantxa, Romero, Adriana, Lio, Pietro, and Bengio, Yoshua. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Wu, Yonghui, Schuster, Mike, Chen, Zhifeng, Le, Quoc V, Norouzi, Mohammad, Macherey, Wolfgang, Krikun, Maxim, Cao, Yuan, Gao, Qin, Macherey, Klaus, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Yang, Zhilin, Cohen, William W, and Salakhutdinov, Ruslan. Revisiting semi-supervised learning with graph embeddings. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning-Volume 48*, pp. 40–48. JMLR. org, 2016.

Ying, Rex, You, Jiaxuan, Morris, Christopher, Ren, Xiang, Hamilton, William L, and Leskovec, Jure. Hierarchical graph representation learning withdifferentiable pooling. *arXiv preprint arXiv:1806.08804*, 2018.

Zhang, Muhan, Cui, Zhicheng, Neumann, Marion, and Chen, Yixin. An end-to-end deep learning architecture for graph classification. In *Proceedings of AAAI Conference on Artificial Intelligence*, 2018.

Zhou, Denny, Bousquet, Olivier, Lal, Thomas N, Weston, Jason, and Schölkopf, Bernhard. Learning with local and global consistency. In *Advances in neural information processing systems*, pp. 321–328, 2004.