Graphs, Convolutions, and Neural Networks

Fernando Gama, Elvin Isufi, Geert Leus, and Alejandro Ribeiro

Abstract

Network data can be conveniently modeled as a graph signal, where data values are assigned to nodes of a graph that describes the underlying network topology. Successful learning from network data is built upon methods that effectively exploit this graph structure. In this work, we overview graph convolutional filters, which are linear, local and distributed operations that adequately leverage the graph structure. We then discuss graph neural networks (GNNs), built upon graph convolutional filters, that have been shown to be powerful nonlinear learning architectures. We show that GNNs are permutation equivariant and stable to changes in the underlying graph topology, allowing them to scale and transfer. We also introduce GNN extensions using edge-varying and autoregressive moving average graph filters, and discuss their properties. Finally, we study the use of GNNs in learning decentralized controllers for robot swarm and in addressing the recommender system problem.

Index Terms

Graph signal processing, graph filters, graph convolutions, graph neural networks, stability

I. Introduction

Data generated by networks are increasingly common in power grids, robotics, biological, social and economic networks, and recommender systems among others. The irregular and complex nature of these network data poses unique challenges, therefore, making successful learning possible only by incorporating the structure into the inner-working mechanisms of the model [1].

Work in this paper is supported by NSF CCF 1717120, ARO W911NF1710438, ARL DCIST CRA W911NF-17-2-0181, ISTC-WAS and Intel DevCloud. F. Gama, and A. Ribeiro are with the Dept. of Electrical and Systems Eng., Univ. of Pennsylvania, USA. E. Isufi is with the Multimedia Computing Group and G. Leus is with the Circuits and Systems Group, Delft Univ. of Technology, The Netherlands. E-mails: {fgama, aribeiro}@seas.upenn.edu, {e.isufi-1, g.j.t.leus}@tudelft.nl.

Convolutional neural networks (CNNs) have epitomized the success of leveraging data structure in temporal series and images transforming the landscape of machine learning in the last decade [2]. CNNs exploit temporal or spatial convolutions to learn an effective nonlinear mapping, scale to large settings, and avoid overfitting [2, Chapter 10]. CNNs offer also some degree of mathematical tractability, allowing to derive theoretical performance bounds under domain perturbations [3]. However, convolutions can only be applied to data in regular domains, hence making CNNs ineffective models when learning from irregular network data.

Graphs are a powerful mathematical tool to model networks and their complex interactions, whereas the network data can be seen as a signal on top of this graph. In recommender systems, for instance, users can be modeled as nodes, their similarities as edges, and the ratings given to items as graph signals. Processing such data by accounting also for the underlying network structure has been the goal of the field of *graph signal processing* (GSP) [1]. GSP has extended the concepts of Fourier transform, graph convolutions, and graph filtering to process signals while accounting for the underlying topology.

Graph convolutional neural networks (GCNNs) build upon graph convolutions to efficiently incorporate the graph structure into the learning process [4]–[6]. GCNNs consist also of a concatenation of layers, in which each layer applies a *graph convolution* followed by a pointwise nonlinearity. GCNNs exhibit the key properties of permutation equivariance and stability to perturbations [7]. The former means GCNNs exploit topological symmetries in the underlying graph, while the latter implies the output is robust to small changes in the graph structure. These results allow GCNNs to scale and transfer, so that they can be trained in one network and then deployed into another similar network, guaranteeing a certain level of performance.

Graph convolutions can be exactly modeled by finite impulse response (FIR) graph filters [8]. But FIR graph filters often require large orders to yield highly discriminatory models, demanding more parameters and an increased computational cost. These limitations are well-understood in the field of GSP and alternative graph filters such as the autoregressive moving average (ARMA) and edge varying graph filters have been proposed to overcome them [9], [10]. ARMA graph filters maintain the full convolutional structure but can achieve a similar performance with fewer parameters. Contrarily, the edge varying graph filters are inspired by their time varying counterparts and adapt the convolutional structure to the specific graph location. The enhanced

flexibility of edge varying graph filters comes at the cost of more parameters but they lay the foundation of a foundational framework for all possible *graph neural networks* (GNNs) [11].

In this overview paper, we give the main ingredients of GCNNs, highlight their permutation equivariance and stability properties, and show how designing a GCNN architecture translates into no more than designing the right graph filter for the application at hand. Section II formally introduces graph convolutions. Section III presents the graph convolutional neural network and discusses permutation equivariance (Sec. III-A) and stability to graph perturbations (Sec. III-B). Section IV generalizes GCNNs by employing alternative graph filters. Section V provides two applications in learning decentralized controllers for flocking a robot swarm and rating prediction in recommender systems. Section VI contains the paper conclusions and future directions.

II. Graphs and Convolutions

We capture the irregular structure of the data by means of an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V} = \{1, \ldots, N\}$ and edge set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The neighborhood of node $i \in \mathcal{V}$ is the set of nodes that share an edge with node i and it is denoted as $\mathcal{N}_i = \{j \in \mathcal{V} : (j,i) \in \mathcal{E}\}$. An $N \times N$ real symmetric matrix $\mathbf{S}$, known as the *graph shift operator*, is associated to the graph, whose (i,j)th entry satisfies $[\mathbf{S}]_{ij} = s_{ij} = 0$ if $(j,i) \notin \mathcal{E}$ for $j \neq i$, i.e., the shift operator has a zero whenever two nodes are disconnected. Common shift operators include the adjacency, Laplacian, and Markov matrices as well as their normalized counterparts [1]. The data on top of this graph forms a graph signal $\mathbf{x} \in \mathbb{R}^N$, where ith entry $[\mathbf{x}]_i = x_i$ is the datum of node i. Entries x_i and x_j are pairwise related to each other if there exists an edge $(i,j) \in \mathcal{E}$. The graph signal $\mathbf{x}$ can be *shifted* over the nodes by the shift operator $\mathbf{S}$ and the ith entry of the shifted signal $\mathbf{Sx}$ is

$$\big[\mathbf{Sx}\big]_i = \sum_{j=1}^{N} [\mathbf{S}]_{ij} [\mathbf{x}]_j = \sum_{j \in \mathcal{N}_i} s_{ij} x_j. \tag{1}$$

where the last equality holds due to the sparsity and locality of $\mathbf{S}$. The shifted signal $\mathbf{Sx}$ is another graph signal where the value at each node is the linear combination of the values of $\mathbf{x}$ at neighboring nodes.

Equipped with the notion of signal shift, we define the *graph convolution* as a linear shift-and-sum operation of a graph signal. Given a set of parameters $\mathbf{h} = [h_0, \ldots, h_K]^{\mathsf{T}}$, the graph

convolution is computed as

$$\mathbf{h} *_{\mathbf{S}} \mathbf{x} = \sum_{k=0}^{K} h_k \ \mathbf{S}^k \mathbf{x} = \mathbf{H}(\mathbf{S})\mathbf{x}. \tag{2}$$

Operation (2) linearly combines the information contained in different neighborhoods. The k-shifted graph signal $\mathbf{S}^k\mathbf{x}$ contains a summary of the information located in the k-hop neighborhood and h_k weighs this summary. This is also a *local* operation since $\mathbf{S}^k\mathbf{x} = \mathbf{S}(\mathbf{S}^{k-1}\mathbf{x})$ entails k repeated information exchanges with one-hop neighbors [cf. (1)]. The graph convolution (2) is analogous to the application of a FIR filter; thus, we refer to the weights h_k as the *filter taps* and to $\mathbf{H}(\mathbf{S})$ as the *filter*. Graph convolutions can therefore be implemented by filtering a graph signal $\mathbf{x}$ with a filter $\mathbf{H}(\mathbf{S})$ as per (2).

We can gain additional insight about graph convolutions by analyzing (2) in the *graph frequency domain*. Consider the eigendecomposition of the shift operator $\mathbf{S} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{\mathsf{T}}$ with orthogonal eigenvector matrix $\mathbf{V} \in \mathbb{R}^{N\times N}$ and diagonal eigenvalue matrix $\mathbf{\Lambda} \in \mathbb{R}^{N\times N}$ ordered as $\lambda_1 \leq \cdots \leq \lambda_N$. The eigenvectors $\mathbf{v}_i$ conform the *graph frequency basis* of graph $\mathcal{G}$ and can be interpreted as signals representing the *graph oscillating modes*, while the eigenvalues λ_i can be considered as *graph frequencies*. We can therefore express any signal $\mathbf{x}$ in terms of these graph oscillating modes as

$$\tilde{\mathbf{x}} = \mathbf{V}^{\mathsf{T}}\mathbf{x}. \tag{3}$$

Operation (3) is known as the *graph Fourier transform* (GFT) of $\mathbf{x}$, in which entry $[\tilde{\mathbf{x}}]_i = \tilde{x}_i$ denotes the *Fourier coefficient* associated to graph frequency λ_i and quantifies the contribution of mode $\mathbf{v}_i$ to the signal $\mathbf{x}$ [1]. Applying the GFT to both sides of (2), yields the input-output frequency relationship

$$\tilde{\mathbf{y}} = \mathbf{V}^{\mathsf{T}}\mathbf{y} = \sum_{k=0}^{K} h_k \mathbf{V}^{\mathsf{T}}\mathbf{V}\ \mathbf{\Lambda}\ \mathbf{V}^{\mathsf{T}}\mathbf{x} = \sum_{k=0}^{K} h_k \mathbf{\Lambda}\ \tilde{\mathbf{x}} = \mathbf{H}(\mathbf{\Lambda})\tilde{\mathbf{x}} \tag{4}$$

where $\mathbf{H}(\mathbf{\Lambda})$ is a diagonal matrix with ith diagonal element $h(\lambda_i)$ for

$$h(\lambda) = \sum_{k=0}^{K} h_k \lambda^k. \tag{5}$$

The function in (5) is the analytic *frequency response* of the graph filter $\mathbf{H}(\mathbf{S})$ and it is determined solely by the filter taps $\mathbf{h}$. The effect that a filter has on a graph signal depends also on the

specific graph through the instantiation of $h(\lambda)$ on the eigenvalues λ_i of $\mathbf{S}$. More precisely, the ith frequency content $\tilde{y}_i$ of the graph convolution output $\mathbf{y}$ is given by

$$\tilde{y}_i = h(\lambda_i)\tilde{x}_i. \tag{6}$$

That is, the graph convolution (2) modifies the ith frequency content $\tilde{x}_i$ of the input signal $\mathbf{x}$ according to the filter value $h(\lambda_i)$ at frequency λ_i. Notice the graph convolution is a pointwise operator in the graph frequency domain, in analogy to the convolution in time and images.

III. Graph Convolutional Neural Networks

Learning from graph data requires identifying a map $\mathbf{\Phi}(\cdot)$ between the data $\mathbf{x}$ and the target representation $\mathbf{y}$ that leverages the graph structure, $\mathbf{y} = \mathbf{\Phi}(\mathbf{x}; \mathbf{S})$. The most elementary of such mappings is the graph convolution $\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H}) = \mathbf{H}(\mathbf{S})\mathbf{x}$ in (2), where set $\mathcal{H} = \{\mathbf{h}\}$ contains the filter coefficients. To *learn* this map, we consider a cost function $J(\cdot)$ and a training set $\mathcal{T} = \{\mathbf{x}_1, \ldots, \mathbf{x}_{|\mathcal{T}|}\}$ with $|\mathcal{T}|$ samples. The *learned map* is then $\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H}^\star)$ with optimal parameters

$$\mathcal{H}^\star = \operatorname*{argmin}_{\mathcal{H}} \frac{1}{|\mathcal{T}|} \sum_{\mathbf{x} \in \mathcal{T}} J\big(\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H})\big). \tag{7}$$

Problem (7) consists of finding the $K+1$ filter taps $\mathcal{H}^\star = \{\mathbf{h}^\star\}$ that best fit the training data w.r.t. cost $J(\cdot)$. However, graph convolutions limit the descriptive power to linear mappings. We can increase the class of mappings that leverage the graph by nesting convolutions into a nonlinearity. The latter leads to the concept of *graph perceptron*, which is formalized next.

Definition 1 (Graph perceptron)**.** A graph perceptron is a mapping that applies an entrywise nonlinearity $\sigma(\cdot)$ to the output of a graph convolution $\mathbf{H}(\mathbf{S})\mathbf{x}$, i.e.,

$$\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H}) = \sigma\big(\mathbf{H}(\mathbf{S})\mathbf{x}\big), \tag{8}$$

where set $\mathcal{H} = \{\mathbf{h}\}$ contains the filter coefficients.

The graph perceptron generates another graph signal obtained as a graph convolution followed by a nonlinearity (e.g., ReLU). As such, the graph perceptron allows capturing nonlinear relationships between the data $\mathbf{x}$ and the target representation $\mathbf{y}$. By building then a cascade of L

graph perceptrons, we get a *multi-layer graph perceptron*, where at layer ℓ we compute

$$\mathbf{x}_\ell = \sigma\big(\mathbf{H}_\ell(\mathbf{S})\mathbf{x}_{\ell-1}\big), \quad \ell = 1, \ldots, L. \tag{9}$$

Differently from (8), a multi-layer graph perceptron allows nonlinear signal mixing between nodes. This can be seen in (9) where the input of the perceptron at layer ℓ is $\mathbf{x}_{\ell-1}$, which is in turn the output of the perceptron at layer $\ell-1$, $\mathbf{x}_{\ell-1} = \sigma(\mathbf{H}_{\ell-1}(\mathbf{S})\mathbf{x}_{\ell-2})$. The cascade form allows, therefore, graph convolutions of nonlinear signal transformations coming from the precedent layer. Unrolling this recursion to all layers, we have that the input to the first layer is the data $\mathbf{x}_0 = \mathbf{x}$ and the output of the last layer is the estimate of the target representation $\mathbf{x}_L = \mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H})$; here, the set $\mathcal{H} = \{\mathbf{h}_\ell\}_\ell$ contains the filter taps of the L graph filters in (9).

The graph perceptron (8) and the multi-layer graph perceptron (9) can be viewed as specific *graph convolutional neural networks* (GCNNs). The former is a GCNN of one layer, while the latter is a GCNN of L layers. As it is a good practice in neural networks [2], [12], we can substantially increase the descriptive power of GCNNs by incorporating multiple parallel features per layer. These features are the result of processing multiple input features with a parallel bank of graph filters. Let us consider $F_{\ell-1}$ input graph signal features $\mathbf{x}_{\ell-1}^1, \ldots, \mathbf{x}_{\ell-1}^{F_{\ell-1}}$ at layer ℓ. Each input feature $\mathbf{x}_{\ell-1}^g$ for $g = 1, \ldots, F_{\ell-1}$ is processed in parallel by F_ℓ different graph filters of the form (2) to output the F_ℓ convolutional features

$$\mathbf{u}_\ell^{fg} = \mathbf{H}_\ell^{fg}(\mathbf{S})\mathbf{x}_{\ell-1}^g = \sum_{k=0}^{K} h_{\ell k}^{fg} \mathbf{S}^k \mathbf{x}_{\ell-1}^g, \quad f = 1, \ldots, F_\ell. \tag{10}$$

The convolutional features are subsequently summarized along the input index g to yield the aggregated features

$$\mathbf{u}_\ell^f = \sum_{g=1}^{F_{\ell-1}} \mathbf{H}_\ell^{fg}(\mathbf{S})\mathbf{x}_{\ell-1}^g, \quad f = 1, \ldots, F_\ell. \tag{11}$$

The aggregated features are finally passed through a nonlinearity to complete the ℓth layer output

$$\mathbf{x}_\ell^f = \sigma(\mathbf{u}_\ell^f), \quad f = 1, \ldots, F_\ell. \tag{12}$$

A GCNN in its complete form is a concatenation of L layers, in which each layer computes operations (10)-(11)-(12). Differently from the multi-layer graph perceptron GCNN in (9), the

complete form employs a parallel bank of $F_\ell \times F_{\ell-1}$ graph convolutional filters. This increases the expressive power of the mapping and exploits both the stable operation in signal processing, the *convolution*, and the underlying graph structure of the data. The input to the first layer is the data $\mathbf{x}_0 = \mathbf{x}$ and the target representation is the collection of F_L features of the last layer $[\mathbf{x}_L^1, \ldots, \mathbf{x}_L^{F_L}] = \mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H})$, where set $\mathcal{H} = \{\mathbf{h}_\ell^{fg}\}_{\ell fg}$ collects now the filter taps of all layers.

We can *learn* the filter taps by solving problem (7) by standard backpropagation with the GCNN map $\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H})$ and set of parameters $\mathcal{H}$ [13]. Since the training data come from graph $\mathbf{S}$, it is expected the learned map $\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H}^\star)$ will *generalize* and perform well for data $\mathbf{x} \notin \mathcal{T}$ that come from the same graph $\mathbf{S}$. The rationale behind this expectation is that the GCNN is a nonlinear processing architecture that exploits the knowledge the graph carries about the data. Another advantage of a GCNN is its local implementation due to the use of graph convolutions [cf. (2)] and pointwise nonlinearities. In fact, all the $F_\ell \times F_{\ell-1}$ convolutional features in (10) are local over the graph as they simply comprise a parallel bank of graph convolutional filters, each of which is local [cf. (2)]. Further, since the aggregation step in (11) happens across features of the same node and the nonlinearity in (12) is pointwise, these operations are also local and distributable. This built-in characteristic of GCNNs naturally leads to learning solutions that are readily distributed on the underlying graph.

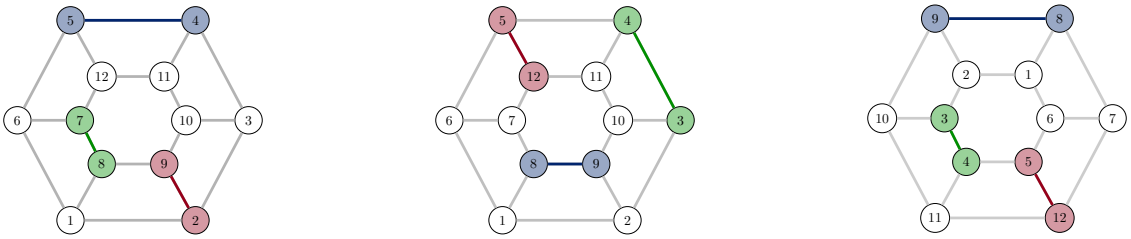
A. Permutation equivariance

A graph shift operator $\mathbf{S}$ fixes an arbitrary ordering of the nodes in the graph. Since nodes are naturally unordered, we want the GCNN output to be unaffected by it. That is, we want any change in node ordering to be reflected with the corresponding reordering in the GCNN output. It turns out GCNNs are unaffected by node labeling –a property known as permutation equivariance– as stated by the following theorem.

Theorem 1 (Permutation Equivariance [7])**.** Consider an $N \times N$ permutation matrix $\mathbf{P}$ and the permutations of the shift operator $\hat{\mathbf{S}} = \mathbf{P}^\mathsf{T}\mathbf{S}\mathbf{P}$ and of the input data $\hat{\mathbf{x}} = \mathbf{P}^\mathsf{T}\mathbf{x}$. For a GCNN $\mathbf{\Phi}(\cdot)$, it holds that

$$\mathbf{\Phi}(\hat{\mathbf{x}}; \hat{\mathbf{S}}, \mathcal{H}) = \mathbf{P}^\mathsf{T}\mathbf{\Phi}(\mathbf{x}; \mathbf{S}, \mathcal{H}). \tag{13}$$

Theorem 1 states that a node reordering results in a corresponding reordering of the GCNN output, implying GCNNs are independent of node labeling. Theorem 1 implies also that graph



(a) Graph $\mathbf{S}$ and signal $\mathbf{x}$ (b) Graph $\mathbf{S}$ and signal $\hat{\mathbf{x}}$ (c) Graph $\hat{\mathbf{S}}$ and signal $\hat{\mathbf{x}}$

Figure 1. Permutation equivariance of GCNNs. The output of a GCNN is equivariant to graph permutations (Theorem 1). This means independence from labeling and shows GCNNs exploit internal signal symmetries. Signals in (a) and (b) are different on the same graph but they are permutations of each other –interchange inner and outer hexagons and rotate $180°$ [c.f. (c)]. A GCNN would learn how to classify the signal in (b) from seeing examples of the signal in (a). Integers represent labels, while colors signal values.

convolutions exploit the inherent symmetries present in a graph to improve data processing. If the graph exhibits several nodes with the same topological neighborhood (graph symmetries), then learning how to process data in any of these nodes can be translated to every other node with the same topological neighborhood. This allows GCNNs to learn from fewer samples and generalize easier to signals located at any topologically similar neighborhood, see Figure 1.

B. Stability to perturbations

Interesting changes in the underlying graph are often more general than permutations. For instance, we might not have access to the actual graph $\mathbf{S}$ but to an estimate $\hat{\mathbf{S}}$ of it, and we still want the GCNN to be reliable. Or we want to train the GCNN on one graph $\mathbf{S}$ but deploy it on another graph $\hat{\mathbf{S}}$ (a scenario known as *transfer learning*). In these cases, we need the GCNN to have a similar performance whether they run on $\mathbf{S}$ or on $\hat{\mathbf{S}}$ as long as both graphs are similar. To measure the similarity between graphs $\mathbf{S}$ and $\hat{\mathbf{S}}$, and in light of Theorem 1, we define next the relative distance modulo permutation.

Definition 2 (Relative distance)**.** Consider the set of all permutation matrices $\mathcal{P} = \{\mathbf{P} \in \{0,1\}^{N \times N} : \mathbf{P}^{\mathsf{T}}\mathbf{1} = \mathbf{1}, \mathbf{P}\mathbf{1} = \mathbf{1}\}$. For two graphs $\mathbf{S}$ and $\hat{\mathbf{S}}$, we define the set of *relative error matrices* as

$$\mathcal{E}(\mathbf{S}, \hat{\mathbf{S}}) = \{\mathbf{E} : \mathbf{P}^{\mathsf{T}}\hat{\mathbf{S}}\mathbf{P} = \mathbf{S} + (\mathbf{E}\mathbf{S} + \mathbf{S}\mathbf{E}) \ , \ \mathbf{P} \in \mathcal{P}\}. \tag{14}$$

The *relative distance* modulo permutations between $\mathbf{S}$ and $\hat{\mathbf{S}}$ is then defined as

$$d(\mathbf{S}, \hat{\mathbf{S}}) = \min_{\mathbf{E} \in \mathcal{E}(\mathbf{S}, \hat{\mathbf{S}})} \|\mathbf{E}\| \tag{15}$$

where $\|\cdot\|$ indicates the operator norm. We denote by $\mathbf{E}^\star$ and $\mathbf{P}^\star$ the relative error matrix and the permutation matrix that minimize (15), respectively.

We readily see that if $d(\mathbf{S},\hat{\mathbf{S}})=0$, then $\hat{\mathbf{S}}$ is a permutation of $\mathbf{S}$, and thus the relative distance (15) measures how far $\mathbf{S}$ and $\hat{\mathbf{S}}$ from being permutations of each other.

The change in the output of a GCNN due to a change in the underlying support is bounded for GCNNs whose constitutive graph filters are integral Lipschitz.

Definition 3 (Integral Lipschitz filters)**.** We say a filter $\mathbf{H}(\mathbf{S})$ is *integral Lipschitz* if its frequency response $h(\lambda)$ [cf. (5)] is such that $|h(\lambda)|\leq 1$ and its derivative $h'(\lambda)$ satisfies $|\lambda h'(\lambda)|\leq C$ for some finite constant C.

The derivative condition $|\lambda h'(\lambda)|\leq C$ implies integral Lipschitz filters have frequency responses that can vary rapidly around $\lambda=0$ but are flat for $\lambda\to\infty$, see Figure 2a.

GCNNs that use integral Lipschitz filters are stable under relative perturbations. This means the change in the GCNN output due to changes in the underlying graph is bounded by the size of the perturbation [cf. Def. 2].

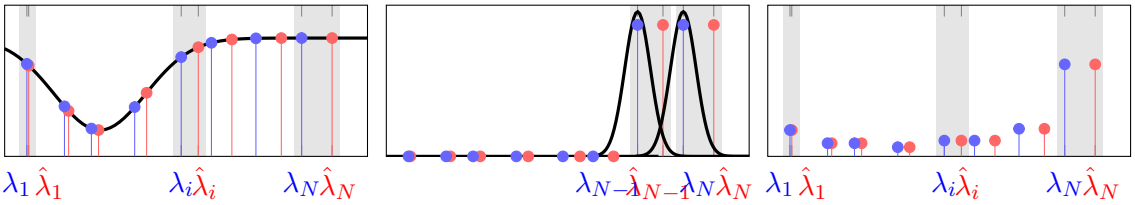
Theorem 2 (Stability [7])**.** Let $\mathbf{S}$ and $\hat{\mathbf{S}}$ be two different graphs such that their relative distance is $d(\mathbf{S},\hat{\mathbf{S}})\leq\varepsilon$ [cf. Def. 2]. Let $\mathbf{\Phi}(\cdot;\cdot,\mathcal{H})$ be a multi-layer graph perceptron GCNN [cf. (9)] where all filters $\mathcal{H}$ are integral Lipschitz with constant C [cf. Def. 3]. Then, it holds that

$$\|\mathbf{\Phi}(\mathbf{x};\mathbf{S},\mathcal{H})-\mathbf{\Phi}(\mathbf{P}^{\star^{\mathsf{T}}}\mathbf{x};\mathbf{P}^{\star^{\mathsf{T}}}\hat{\mathbf{S}}\mathbf{P}^\star,\mathcal{H})\|\leq 2C\ (1+\delta\sqrt{N})\ L\ \varepsilon\ \|\mathbf{x}\|+\mathcal{O}(\varepsilon^2) \tag{16}$$

where $\delta=(\|\mathbf{U}-\mathbf{V}\|+1)^2-1$ is the eigenvector misalignment between the eigenbasis $\mathbf{V}$ of $\mathbf{S}$ and the eigenbasis $\mathbf{U}$ of the relative error matrix $\mathbf{E}^\star$, with $\mathbf{E}^\star$ and $\mathbf{P}^\star$ given in Definition 2.

Theorem 2 proves that a change ε in the shift operator causes a change proportional to ε in the GCNN output. The proportionality constant has the term C that depends on the filter design, and the term $(1+\delta\sqrt{N})$ that depends on the specific perturbation. But it also has a constant factor L that depends on the depth of the architecture implying deeper GCNNs are less stable.

To offer further insight into Theorem 2, consider the perturbation $\hat{\mathbf{S}}$ to be an edge dilation of a graph $\mathbf{S}$, i.e. $\hat{\mathbf{S}}=(1+\varepsilon)\mathbf{S}$, where all edges are increased proportionally by a factor ε. The relative error matrix is $\mathbf{E}=(\varepsilon/2)\mathbf{I}$ so that the relative distance is $d(\mathbf{S},\hat{\mathbf{S}})=\|\mathbf{E}\|\leq\varepsilon$. The graph



(a) Integral Lipschitz Filter (b) High eigenvalue features (c) Frequency mixing

Figure 2. (a) Frequency response for an integral Lipschitz filter (in black), eigenvalues for $\mathbf{S}$ (in blue) and eigenvalues for $\hat{\mathbf{S}}$ (in red). Larger eigenvalues exhibit a larger change. (b) Separating energy located at λ_{N-1} from that at λ_N requires filters with sharp transitions that are not integral Lipschitz. Then, a change in eigenvalues renders these filters useless (they are not stable) (c) Applying a ReLU to a signal with all its energy located at λ_N results in a signal with energy spread through the spectrum. Information on low eigenvalues can be discriminated in a stable fashion.

dilation changes the eigenvalues to $\hat{\lambda}_i = (1+\varepsilon)\lambda_i$ while the eigenvectors remain the same. We note that, even if ε is small, the change in eigenvalues could be large if λ_i is large, see Figure 2a.

This observation that even small perturbations lead to large changes in the eigenvalues can considerably affect the output of a graph filter causing instability, unless the graph filters are carefully designed. To see this, consider first the output of a graph filter in the frequency domain, $\tilde{y}_i = h(\lambda_i)\tilde{x}_i$ [cf. (6)]. With the graph dilation, the frequency response gets instantiated at $\hat{\lambda}_i = (1+\varepsilon)\lambda_i$ instead of λ_i, so the ith frequency content is now $\hat{\tilde{y}}_i = h(\hat{\lambda}_i)\tilde{x}_i$. The change between the original ith frequency content of the output $\tilde{y}_i$ and the perturbed one $\hat{\tilde{y}}_i$ depends on how much $h(\lambda_i)$ changes with respect to $h(\hat{\lambda}_i)$, and thus can be quite large for large λ. So if we want $\tilde{y}_i$ to be close to $\hat{\tilde{y}}_i$ for stability, we need to have frequency responses $h(\lambda)$ that have a flat response for large eigenvalues, see Figure 2a. Integral Lipschitz filters do have a flat response at large eigenvalues and thus are stable.

The cost to pay for stability is that integral Lipschitz filters are not able to discriminate information located at higher eigenvalues. As seen in Figure 2b, discriminative filters are narrow filters. Then, if even a small perturbation causes a large change in the instantiated eigenvalue (as is the case for large eigenvalues), the filter output changes to a zero output, and thus is not stable. In essence, linear filters are either stable or discriminative, but cannot be both.

GCNNs incorporate pointwise nonlinearities in the graph perceptron. This nonlinear operation has a frequency mixing effect (akin to demodulation) by which the signal energy is spilled throughout the spectrum, see Figure 2c. Thus, energy from large eigenvalues now appears in smaller eigenvalues. This new low-eigenvalue frequency content can be captured and discrimi-

nated by subsequent filters in a stable manner. Therefore, pointwise nonlinearities make GCNNs information processing architectures that are both stable and selective.

IV. Extensions: General Graph Filters

Oftentimes, the GCNN would require highly sharp filter responses to discriminate between classes. We can increase the discriminatory power by either increasing the filter order K or changing the filter type $\mathbf{H}(\mathbf{S})$ in the graph perceptron (8). Increasing K is not always feasible as it leads to more filter coefficients, a higher complexity, and numerical issues related to the higher order powers of the shift operator $\mathbf{S}^k$. Instead, changing the filter type allows implementing another family of graph neural networks (GNNs) with different properties. We present in the sequel two alternative filters that provide different insights on how to design more general GNNs: *the autoregressive moving average* (ARMA) graph filter [9] and *the edge varying* graph filter [10].

A. ARMANet

An ARMA graph filter operates also pointwise in the spectrum domain $\tilde{y}_i = h(\lambda_i)\tilde{x}_i$ [cf. (6)] but it is characterized by the rational frequency response

$$h(\lambda) = \frac{\sum_{q=0}^{Q} b_q \lambda^q}{1 + \sum_{p=1}^{P} a_p \lambda^p}. \tag{17}$$

The frequency response is now controlled by P denominator coefficients $\mathbf{a} = [a_1, \ldots, a_P]^\mathsf{T}$ and $Q+1$ numerator coefficients $\mathbf{b} = [b_0, \ldots, b_Q]^\mathsf{T}$. The rational frequency responses in (17) span an equivalent space to that of graph filters in (2). However, the spectral equivalence does not imply that the two filters have the same properties. Our expectation is that for the same number of filter coefficients, ARMA filters will produce more discriminative frequency responses compared with the FIR filters in (5). Replacing the spectral variable λ with the shift operator variable $\mathbf{S}$ allows us to write the ARMA output $\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x}$ as

$$\mathbf{y} = \Big(\mathbf{I} + \sum_{p=1}^{P} a_p \mathbf{S}^p\Big)^{-1} \Big(\sum_{q=0}^{Q} b_q \mathbf{S}^q\Big)\mathbf{x} := \mathbf{P}(\mathbf{S})^{-1}\mathbf{Q}(\mathbf{S})\mathbf{x} \tag{18}$$

where $\mathbf{P}(\mathbf{S}) := \mathbf{I} + \sum_{p=1}^{P} a_p \mathbf{S}^p$ and $\mathbf{Q} := \sum_{q=0}^{Q} b_1 \mathbf{S}^q$ are two FIR filters [cf. (2)] that allow writing the ARMA filter as $\mathbf{H}(\mathbf{S}) = \mathbf{P}(\mathbf{S})^{-1}\mathbf{Q}(\mathbf{S})$. As it follows from (18), we need to apply the matrix inverse $\mathbf{P}(\mathbf{S})$ to obtain the ARMA output. This, unless the number of nodes is

moderate, is computationally unaffordable; hence, we need an iterative method to approximately apply the inverse. Due to its faster convergence, we choose a parallel structure that consists of first transforming the polynomial ratio in (18) in its partial fraction decomposition form and subsequently using the Jacobi method to approximately apply the inverse. While also other Krylov approaches are possible to solve (18), the parallel Jacobi method offers a better tradeoff between computational complexity, distributed implementation, and convergence to the inverse solution.

Partial fraction decomposition of ARMA filters. Consider the rational frequency response $h(\lambda)$ in (17) and let $\boldsymbol{\gamma} = [\gamma_1, \ldots, \gamma_P]^{\mathsf{T}}$ be the P poles, $\boldsymbol{\beta} = [\beta_1, \ldots, \beta_P]^{\mathsf{T}}$ the corresponding residuals and $\boldsymbol{\alpha} = [\alpha_0, \ldots, \alpha_K]^{\mathsf{T}}$ the direct terms. Then, we can write (18) in the equivalent form

$$\mathbf{y} = \sum_{p=1}^{P} \beta_p \Big(\mathbf{S} - \gamma_p \mathbf{I} \Big)^{-1} \mathbf{x} + \sum_{k=0}^{K} \alpha_k \mathbf{S}^k \mathbf{x}. \tag{19}$$

The equivalence of (19) and (18) implies that instead of learning $\mathbf{a}$ and $\mathbf{b}$ in (18), we can learn $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, and $\boldsymbol{\gamma}$ in (19). To avoid the matrix inverses in the single pole filters, we can approximate each output $\mathbf{u}_p$ through the Jacobi method.

Jacobi method for single pole filters. We can write the output of the pth single pole filter $\mathbf{u}_p$ in the equivalent linear equation form $(\mathbf{S} - \gamma_p \mathbf{I})\mathbf{u}_p = \beta_p \mathbf{x}$. The Jacobi algorithm requires separating $(\mathbf{S} - \gamma_p \mathbf{I})$ into its diagonal and off-diagonal terms. Defining $\mathbf{D} = \text{diag}(\mathbf{S})$ as the matrix containing the diagonal of the shift operator, we can write the Jacobi approximation $\mathbf{u}_{p\tau}$ of the pth single pole filter output $\mathbf{u}_p$ at iteration τ by the recursive expression

$$\mathbf{u}_{p\tau} = \Big(\mathbf{D} - \gamma_p \mathbf{I} \Big)^{-1} \Big[\beta_p \mathbf{x} - \Big(\mathbf{S} - \mathbf{D} \Big) \mathbf{u}_{p(\tau-1)} \Big], \qquad \text{with } \mathbf{u}_{p0} = \mathbf{x}. \tag{20}$$

The inverse in (20) is now element-wise on the diagonal matrix $(\mathbf{D} - \gamma_p \mathbf{I})$. This recursion can be unrolled to all its terms to write an explicit relationship between $\mathbf{u}_{p\tau}$ and $\mathbf{x}$. To do that, we define the parameterized shift operator $\mathbf{R}(\gamma_p) = -\big(\mathbf{D} - \gamma_p \mathbf{I}\big)^{-1} \big(\mathbf{S} - \mathbf{D}\big)$ and use it to write the Tth Jacobi recursion as

$$\mathbf{u}_{pT} = \beta_p \sum_{\tau=0}^{T-1} \mathbf{R}^{\tau}(\gamma_p) \mathbf{x} + \mathbf{R}^{T}(\gamma_p) \mathbf{x}. \tag{21}$$

For a convergent Jacobi method, $\mathbf{u}_{pT}$ converges to the single pole output $\mathbf{u}_p$. However, in a practical setting we truncate (21) for a finite T. We can then write the single pole filter output

as $\mathbf{u}_{pT} := \mathbf{H}_T(\mathbf{R}(\gamma))\mathbf{x}$, where we define the following FIR filter of order T

$$\mathbf{H}_T(\mathbf{R}(\gamma_p)) = \beta_p \sum_{\tau=0}^{T-1} \mathbf{R}^\tau(\gamma_p) + \mathbf{R}^T(\gamma_p). \tag{22}$$

with the parametric shift operator $\mathbf{R}(\gamma)$. In other words, a single pole filter is approximated by a graph convolutional filter of the form (2) in which the shift operator $\mathbf{S}$ is substituted by $\mathbf{R}(\gamma)$. This parametric convolutional filter uses coefficients β_p for $\tau = 0, \ldots, T-1$ and 1 for $\tau = T$.

Jacobi ARMA filters and ARMANets. Assuming we use truncated Jacobi iterations of order T to approximate all single pole filters in (19), we can write the ARMA filter as

$$\mathbf{H}(\mathbf{S}) = \sum_{p=1}^{P} \mathbf{H}_T(\mathbf{R}(\gamma_p)) + \sum_{k=0}^{K} \alpha_k \mathbf{S}^k \tag{23}$$

where the pth approximated single pole filter $\mathbf{H}_T(\mathbf{R}(\gamma_p))$ is defined in (22) and the parametric shift operator $\mathbf{R}(\gamma_p)$ in (21). In summary, a Jacobi approximation of the ARMA filter with orders (P, T, K) is the one defined by (22) and (23). Scalar P indicates the number of poles, T the number of Jacobi iterations, and K the order of the direct term $\sum_{k=0}^{K} \alpha_k \mathbf{S}^k$ in (19).

Substituting (23) into (8) yields an ARMA graph perceptron, which is the building block for ARMA GNNs or, for short, ARMANets. ARMANets are themselves convolutional. For a sufficiently large number of Jacobi iterations T, (23) is equivalentnt to (18) which performs a pointwise multiplication in the spectrum domain with the response (17). The Jacobi filters in (23) are also reminiscent of the convolutional filters in (2) as shown by (23). But the similarity is superficial because in ARMANets we train also the $2P$ single pole filter coefficients β_p and γ_P alongside the $K+1$ coefficients of the direct term $\sum_{k=0}^{K} \alpha_k \mathbf{S}^K$. The equivalence suggests ARMANets may help achieving more discriminatory architectures by tuning the single pole filter approximation orders P and T.

B. EdgeNet

While ARMANets enhance the discriminatory power of GCNNs with alternative convolutional filters, the edge varying GNN departs from the convolutional prior to improve GCNNs. The EdgeNet leverages the sparsity and locality of the shift operator $\mathbf{S}$ and forms a graph perceptron [cf. (9)] by replacing the graph convolutional filter with an edge varying graph filter [10].

From shared to edge parameters. In the convolutional filter (2), all nodes share the same scalar h_k to weigh equally the information from all k-hop away neighbors $\mathbf{S}^k\mathbf{x}$. This is advantageous because it limits the number of trainable parameters, allows permutation equivariance, and favors stability. However, this parameter sharing limits also the discriminatory power to architectures whose filters $\mathbf{H}(\mathbf{S})$ have the same eigenvectors as $\mathbf{S}$ [cf. (4)]. We can improve the discriminatory power by considering a linear filter in which node i uses a scalar $\Phi_{ij}^{(k)}$ to weigh the information of its neighbor j at iteration k. For $k = 0$, each node weighs only its own signal to build the zero-shifted signal $\mathbf{z}^{(0)} = \mathbf{\Phi}^{(0)}\mathbf{x}$, where $\mathbf{\Phi}^{(0)}$ is an $N \times N$ diagonal matrix of parameters with ith diagonal entry $\Phi_{ii}^{(0)}$ being the weight of node i. Signal $\mathbf{z}^{(0)}$ is subsequently exchanged with neighboring nodes to build the one-shifted signal $\mathbf{z}^{(1)} = \mathbf{\Phi}^{(1)}\mathbf{z}^{(0)}$, where the parameter matrix $\mathbf{\Phi}^{(1)}$ shares the support with $\mathbf{I} + \mathbf{S}$; the (i, j)th entry $\Phi_{ij}^{(1)}$ is the weight node i applies to signal $z_j^{(0)}$ from neighbor j. Repeating the latter for k shifts, we get the recursion

$$\mathbf{z}^{(k)} = \mathbf{\Phi}^{(k)}\mathbf{z}^{(k-1)} = \prod_{k'=0}^{k} \mathbf{\Phi}^{(k')}\mathbf{x} = \mathbf{\Phi}^{(k:0)}\mathbf{x}, \quad k = 0, \ldots, K \tag{24}$$

where the product matrix $\mathbf{\Phi}^{(k:0)} = \prod_{k'=0}^{k} \mathbf{\Phi}^{(k')} = \mathbf{\Phi}^{(k)} \cdots \mathbf{\Phi}^{(0)}$ accounts for the weighted propagation of the graph signal $\mathbf{z}^{(-1)} = \mathbf{x}$ from at most k-hops away neighbors. Each node is therefore free to adapt its weights for each iteration k to capture the necessary local detail.

Edge varying filters and EdgeNets. The collection of signals $\mathbf{z}^{(k)}$ in (24) behaves like a sequence of parametric shifts, where at iteration k we use the parametric shift operator $\mathbf{\Phi}^{(k)}$ to shift-and-weigh the signal. Following the same idea as in (2), we can sum up *edge varying shifted* signals $\mathbf{z}^{(k)}$ to get the input-output map $\mathbf{y} = \mathbf{H}(\mathbf{\Phi})\mathbf{x}$ of an edge varying graph filter. For this relation to hold, the filter matrix $\mathbf{H}(\mathbf{\Phi})$ should satisfy

$$\mathbf{H}(\mathbf{\Phi}) = \sum_{k=0}^{K} \mathbf{\Phi}^{(k:0)} = \sum_{k=0}^{K} \Big(\prod_{k'=0}^{k} \mathbf{\Phi}^{k'} \Big). \tag{25}$$

The edge varying graph filter is characterized by the $K + 1$ parameter matrices $\mathbf{\Phi}^{(0)}, \ldots, \mathbf{\Phi}^{(K)}$ and contains $K(M+N)+N$ parameters. The edge varying graph filter forms the broadest family of graph filters: it generalizes the FIR filter in (2) (for $\mathbf{\Phi}^{(k:0)} = h_k\mathbf{S}^k$), the ARMA filter in (23), and almost all other filters employed to build GNNs including spectral filters [4], Chebyshev filters [5], Cayley filters, graph isomorphism filters, and also graph attention filters [14].

Substituting (25) into (8) yields an edge varying graph perceptron, which is the building block for edge varying GNNs or, for short, EdgeNets. EdgeNets are more than convolutional architectures; the high number of degrees of freedom and linear complexity render EdgeNets strong candidates for highly discriminatory GNNs in contained graphs. If the graph is large, the EdgeNet can efficiently trade some edge detail (e.g., allowing edge varying weights only to a few nodes) to make the number of parameters independent of the graph dimension [11].

V. Applications

A. Learning Decentralized Controllers for Flocking

The objective of flocking is to coordinate a team of agents to fly together with the same velocity while avoiding collisions. The agents start flying at arbitrary velocities and need to take appropriate actions that would allow them to flock together. This problem has a straightforward *centralized* solution that amounts to setting each agent's velocity to the average velocity of the team. However, *decentralized* solutions are famously difficult to find [15]. Since GCNNs are naturally distributed, we use them to *learn* the decentralized controllers.

Let us consider a set of N agents, in which each agent i is described at discrete time t by its position $\mathbf{r}_i(t) \in \mathbb{R}^2$, velocity $\mathbf{v}_i(t) \in \mathbb{R}^2$ and acceleration $\mathbf{u}_i(t) \in \mathbb{R}^2$. The collision-avoidance flocking problem has the optimal solution on the accelerations

$$\mathbf{u}_i^\star(t) = -\sum_{j=1}^{N} \Big(\mathbf{v}_i(t) - \mathbf{v}_j(t)\Big) - \sum_{j=1}^{N} \rho\big(\mathbf{r}_i(t), \mathbf{r}_j(t)\big) \tag{26}$$

where $\rho(\mathbf{r}_i(t), \mathbf{r}_j(t))$ represents the collision-avoidance potential [16, eq. (10)]. As it follows from (26), agent i requires velocity knowledge from all other agents to compute the optimal solution. In contrast to this *centralized* controller, we would like to use GCNNs to learn a decentralized solution that can be computed only with local exchanges between agents. Two agents i and j can communicate if $\|\mathbf{r}_i(t) - \mathbf{r}_j(t)\| \leq R$ for some communication radius R. The communication network evolves through a succession of graphs $\mathcal{G}(t) = (\mathcal{V}, \mathcal{E}(t))$ where $\mathcal{V}$ is the set of agents, $\mathcal{E}(t)$ the set of edges at time t, and let $\mathbf{S}(t)$ be the associated shift operator.

We use a GCNN to *learn* a decentralized controller $\mathbf{U}(t) = \mathbf{\Phi}(\mathbf{R}(t), \mathbf{V}(t); \mathbf{S}(t), \mathcal{H})$ where $\mathbf{U}(t)$, $\mathbf{R}(t)$ and $\mathbf{V}(t)$ are the $N \times 2$ matrices that collect the accelerations $\mathbf{u}_i(t)$, the velocities $\mathbf{v}_i(t)$ and the positions $\mathbf{r}_i(t)$ of all agents, respectively. The filters involved are adapted to the

Table I: Average (std. deviation) cost for different parameters in flocking. Optimal cost: $51(\pm 1)$.

F / K	1	2	3	1	2	3
16	$763(\pm 307)$	$868(\pm 1097)$	$482(\pm 168)$	$137(\pm 9)$	$129(\pm 4)$	$126(\pm 5)$
32	$563(\pm 335)$	$501(\pm 187)$	$\mathbf{472}(\pm \mathbf{236})$	$80(\pm 3)$	$81(\pm 9)$	$\mathbf{78}(\pm \mathbf{3})$
64	$559(\pm 260)$	$667(\pm 705)$	$500(\pm 257)$	$86(\pm 6)$	$83(\pm 5)$	$83(\pm 4)$
	FIR Filter			GCNN		

delayed nature of the information structure, i.e. $\mathbf{H}(\mathbf{S}(t))\mathbf{x}(t) = \sum_{k=0}^{K} h_k \mathbf{S}(t)\mathbf{S}(t-1)\cdots\mathbf{S}(t-k+1)\mathbf{x}(t-k)$ [cf. (2)]. We use imitation learning [17] to train the GCNN over a set of trajectories $\mathcal{T} = \{(\mathbf{R}(t), \mathbf{V}(t), \mathbf{U}^{\star}(t))_t\}$ where $\mathbf{U}^{\star}(t) \in \mathbb{R}^{N\times 2}$ is the collection of optimal actions for each agent $\mathbf{u}_i^{\star}(t)$, whose closed-form (26) is easy to obtain and required only during training.

Setup. We consider a network of $N = 50$ agents with communication radius $R = 2$m and maximum acceleration $u_{\max} = 10\text{m/s}^2$. To train the network, we generated 400 trajectories for training, 20 for validation, and 20 for test, where each trajectory has a duration of 2s and a sampling time $T_s = 0.01$s. The agents are initially placed at random in a circle with a minimum separation of 0.1m and initial velocity sampled uniformly at random in $v_0 \in [-3, 3]\text{m/s}$. To evaluate the performance, we consider the velocity variance of the team aggregated over time [16, eq. (13)]. The latter is averaged over 10 realizations.

We compare the performance of a single-layer GCNN to that of a parallel bank of FIR graph filters (linear distributed mapping). For both cases, we consider the same number of features F, and filter order K, while the GCNN has an hyperbolic tangent nonlinearity. These architectures are followed by a readout layer in each agent to map the resulting F features into the two-dimensional acceleration $\mathbf{u}_i(t)$. We trained for 40 epochs with batch size of 20 samples using ADAM optimizer with learning rate 5×10^{-4} and forgetting factors 0.9 and 0.999.

Results. In the first experiment, we evaluated the performance of different combinations of $F \in \{16, 32, 64\}$ and $K \in \{1, 2, 3\}$. Table I shows an order of magnitude improvement of the GCNN compared with the linear FIR filter map. Further, the best FIR cost is also $10\times$ worse than the optimal trajectory cost, while the GCNN is only $1.5\times$ worse. Finally, note the FIR filter exhibits also an unreasonable standard deviation, which showcases its poor transferability properties.

In the second experiment, we set $F = 32$ and $K = 3$ and study the scalability of the learned controller. We train the GCNN and the FIR on a network of $N = 50$ agents but test on networks with an increasing number of agents. Notice that changing the size of the graph is straightforward

Table II: Scalability. Trained on 50 agents. Tested on N agents. Optimal cost: $51(\pm 1)$.

N	50	62	75	87	100
FIR filter	$408(\pm 88)$	$408(\pm 93)$	$434(\pm 128)$	$420(\pm 105)$	$430(\pm 131)$
GCNN	$77(\pm 3)$	$78(\pm 3)$	$77(\pm 2)$	$77(\pm 2)$	$78(\pm 2)$

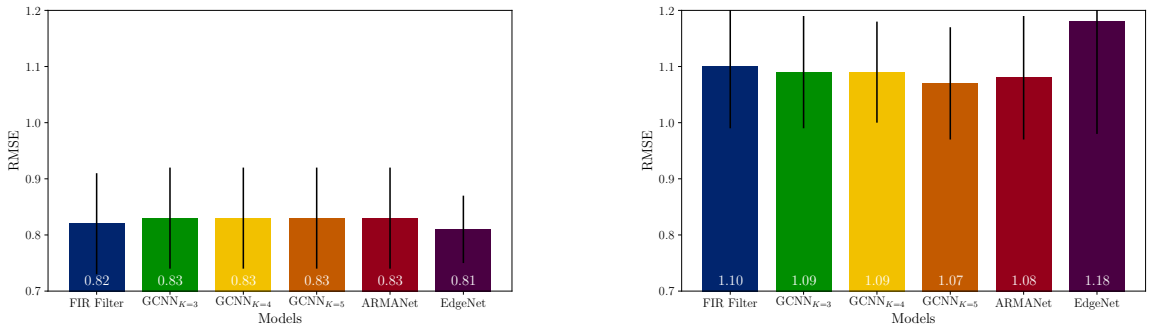
when using graph convolutions, since the learned filters taps $\mathcal{H}$ can act on shift operators of different sizes [cf. (2)]. Table II shows the GCNN learned controller achieve perfect scalability since their performance remains unaltered, therefore, promoting GCNNs as a powerful tool to be trained efficiently on smaller networks and then successfully deployed on larger ones.

B. Recommender Systems

As a second application, we consider rating prediction in recommender systems. In specific, we are interested in estimating the rating a user would give to a specific movie based on ratings given to other movies [18]. Based on the ratings or features, movies exhibit similarity patterns that can be captured by a graph, in which nodes are movies and edge weights indicate similarity strength. The graph signal consists of the ratings a user has given to the movies watched and has missing values on those nodes (movies) not rated. We are therefore interested in using a GNN for interpolating the missing values in a data-driven manner.

Setup. We consider a subset of the MovieLens-100k dataset, containing the 200 movies with the largest number of ratings [19]. The resulting dataset has $47,825$ ratings given by 943 users to some of those 200 movies. The similarity between movies is the Pearson correlation [18, eq. (6)], which is further sparsified to keep only the ten edges with stronger similarity. We split the dataset into 90% for training and 10% for testing. In this context, each user represents a graph signal, where the value at each node is the rating given to that movie. Movies not rated are given a value of 0. The rating of the specific movie of interest is extracted as a label, and zeroed out in the graph signal. We run 10 random dataset split and measure the performance through the root mean squared error.

We compare six different single-layer architectures. A FIR graph filter of order $K = 4$ [18]; three GCNNs of orders $K = 3$, $K = 4$ and $K = 5$, respectively; an ARMANet of order $P = 4$ and $T = 1$ Jacobi iteration [cf. (23)]; and an EdgeNet of order $K = 4$ [cf. (25)]. The number of features in all cases is $F = 64$, and ReLU nonlinearities are used. We follow each architecture



(a) Train and test on same movie

(b) Train on one movie, and test on another

Figure 3. RMSE and standard deviation for different architectures in the movie recommendation problem.

with a readout layer (local to each node) that maps the F output features into the single scalar estimated rating. The FIR filter serves as benchmark, since it has shown superior performance compared with traditional collaborative filtering in [18]. We train the architectures for 40 epochs with batch size of five samples, using an ADAM optimizer with learning rate 5×10^{-3} and forgetting factors 0.9 and 0.999.

Results. In the first experiment, we estimate the ratings for the movie with the largest number of ratings (*Star Wars*). Figure 3a shows the EdgeNet performs the best with an RMSE of $0.81(\pm 0.05)$, followed closely by all the other architectures, $0.83(\pm 0.09)$. This can be explained by the increased expressive power EdgeNets have. In the second experiment, we focus on transferability of the GNNs —a consequence of the stability Theorem 2. We considered the trained mappings in *Star Wars* and used them to estimate the ratings of the movie with the second-largest number of ratings (*Contact*). Figure 3b shows the GCNN performs the best with order $K = 5$, yielding an RMSE of $1.07(\pm 0.09)$. The performance of the EdgeNet, on the contrary, has notoriously degraded, resulting in an RMSE of $1.18(\pm 0.20)$. We attribute this degradation to the fact that EdgeNets are not permutation equivariant nor transferable.

VI. Conclusion

Graphs are powerful algebraic tools to model complex networks and represent their data as graph signals. Graph signal processing has established the key ingredients to handle these signals, where by providing a rigorous definition of graph convolutions, it allows learning meaningful linear models that account for the graph structure. Similar to time and images signals, convolutions of graph signals can be implemented by finite impulse response (graph) filters.

Graph convolutional neural networks nest graph filters into a pointwise nonlinearity and bring learning over networks to the next level. This simple but powerful structure allows GCNNs to inherit the local and distributed implementation of graph filters while describing nonlinear behaviors. Graph convolutions allow also providing theoretical insights onto GCNNs behavior such as permutation equivariance and stability to perturbations. Permutation equivariance helps to adequately exploit topological symmetries while stability helps to transfer learning; together they guarantee scalability. These theoretical analysis form the first step towards treating GCNNs as grey-box learning solutions. The graph provides a strong prior about the network and the data. Thus, we can develop an adequate GCNN architecture by choosing the right shift operator to represent said structure and the right graph filter to cope with the data behavior. For instance, ARMA graph filters are a viable convolutional solution to build an architecture with less but shared parameters, which are useful when learning from limited data. Edge varying graph filters instead look for solutions beyond convolutions, forming the most general graph neural network model. Applications in learning decentralized controllers for flocking and rating prediction illustrated the superior performance of GCNNs and their outstanding scalability compared with linear models.

Without any doubt signal processing will keep playing a role in tackling important challenges when processing, modeling, and learning from network data. Graphs, convolutions, and neural networks form a triad that facilitates these tasks by effectively handling inherent nonlinear relationships within the data. However, additional research is needed in GNNs and with GNNs. In GNNs, work needs to be done for developing new filters that capture relevant properties of specific topologies. Work also needs to be done for developing theoretical insights on the GNN performance and its limitations. With GNNs, research is needed to exploit these nonlinear models for distributed learning and control as well as deploying them into power grids, internet-of-things, and biological networks, to name just a few.

REFERENCES

[1] A. Ortega, P. Frossard, J. Kovačević, J. M. F. Moura, and P. Vandergheynst, "Graph signal processing: Overview, challenges and applications," *Proc. IEEE*, vol. 106, no. 5, pp. 808–828, May 2018.

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, ser. The Adaptive Computation and Machine Learning Series. Cambridge, MA: The MIT Press, 2016.

[3] S. Mallat, "Group invariant scattering," *Commun. Pure, Appl. Math.*, vol. 65, no. 10, pp. 1331–1398, Oct. 2012.

[4] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and deep locally connected networks on graphs," in *2nd Int. Conf. Learning Representations*. Banff, AB: Assoc. Comput. Linguistics, 14-16 Apr. 2014, pp. 1–14.

[5] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *30th Conf. Neural Inform. Process. Syst.* Barcelona, Spain: Neural Inform. Process. Foundation, 5-10 Dec. 2016, pp. 3844–3858.

[6] F. Gama, A. G. Marques, G. Leus, and A. Ribeiro, "Convolutional neural network architectures for signals supported on graphs," *IEEE Trans. Signal Process.*, vol. 67, no. 4, pp. 1034–1049, Feb. 2019.

[7] F. Gama, J. Bruna, and A. Ribeiro, "Stability properties of graph neural networks," *arXiv:1905.04497v2 [cs.LG]*, 4 Sep. 2019. [Online]. Available: http://arxiv.org/abs/1905.04497

[8] A. Sandryhaila and J. M. F. Moura, "Big data analysis with signal processing on graphs: Representation processing of massive data sets with irregular structure," *IEEE Signal Process. Mag.*, vol. 31, no. 5, pp. 80–90, Sep. 2014.

[9] E. Isufi, A. Loukas, A. Simonetto, and G. Leus, "Autoregressive moving average graph filtering," *IEEE Trans. Signal Process.*, vol. 65, no. 2, pp. 274–288, Jan. 2017.

[10] M. Coutino, E. Isufi, and G. Leus, "Advances in distributed graph filtering," *IEEE Trans. Signal Process.*, vol. 67, no. 9, pp. 2320–2333, May 2019.

[11] E. Isufi, F. Gama, and A. Ribeiro, "EdgeNets: Edge varying graph neural networks," *arXiv:2001.07620v1 [cs.LG]*, 21 Jan. 2020. [Online]. Available: http://arxiv.org/abs/2001.07620

[12] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, pp. 251–257, 25 Oct. 1991.

[13] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986.

[14] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th Int. Conf. Learning Representations*. Vancouver, BC: Assoc. Comput. Linguistics, 30 Apr.-3 May 2018, pp. 1–12.

[15] H. S. Witsenhausen, "A counterexample in stochastic optimum control," *SIAM J. Control*, vol. 6, no. 1, pp. 131–147, 1968.

[16] E. Tolstaya, F. Gama, J. Paulos, G. Pappas, V. Kumar, and A. Ribeiro, "Learning decentralized controllers for robot swarms with graph neural networks," in *Conf. Robot Learning 2019*. Osaka, Japan: Int. Found. Robotics Res., 30 Oct.-1 Nov. 2019.

[17] S. Ross and D. Bagnell, "Efficient reductions for imitation learning," in *13th Int. Conf. Artificial Intell., Statist.* Sardinia, Italy: Proc. Mach. Learning Res., 13-15 May 2010, pp. 661–668.

[18] W. Huang, A. G. Marques, and A. Ribeiro, "Rating prediction via graph signal processing," *IEEE Trans. Signal Process.*, vol. 66, no. 19, pp. 5066–5081, Oct. 2018.

[19] F. M. Harper and J. A. Konstan, "The MovieLens datasets: History and context," *ACM Trans. Interactive Intell. Syst.*, vol. 5, no. 4, pp. 19:(1–19), Jan. 2016.