Improving Graph Convolutional Networks with Non-Parametric Activation Functions

Simone Scardapane*, Steven Van Vaerenbergh†, Danilo Comminiello* and Aurelio Uncini*
*Dept. Information Eng., Electronics and Telecomm., Sapienza University of Rome, Italy.
†Dept. Communications Eng., University of Cantabria, Spain.
Corresponding author email: simone.scardapane@uniroma1.it

***Abstract*—Graph neural networks (GNNs) are a class of neural networks that allow to efficiently perform inference on data that is associated to a graph structure, such as, e.g., citation networks or knowledge graphs. While several variants of GNNs have been proposed, they only consider simple nonlinear activation functions in their layers, such as rectifiers or squashing functions. In this paper, we investigate the use of graph convolutional networks (GCNs) when combined with more complex activation functions, able to adapt from the training data. More specifically, we extend the recently proposed kernel activation function, a non-parametric model which can be implemented easily, can be regularized with standard ℓ_p-norms techniques, and is smooth over its entire domain. Our experimental evaluation shows that the proposed architecture can significantly improve over its baseline, while similar improvements cannot be obtained by simply increasing the depth or size of the original GCN.**

I. Introduction

Efficient processing of graph-structured data (e.g., citation networks) has a range of different applications, going from bioinformatics to text analysis and sensor networks, among others. Of particular importance is the design of learning methods that are able to take into account both numerical characteristics of each node in the graph and their inter-connections [1]. While graph machine learning techniques have a long history, recently we witness a renewed interest in models that are defined and operate directly in the graph domain (as compared to a Euclidean space), instead of including the graph information only a posteriori in the optimization process (e.g., through manifold regularization techniques [2]), or in a pre-processing phase via graph embeddings [3]. Examples of native graph models include graph linear filters [4], their kernel counterparts [5], and graph neural networks (GNNs) [6], [7].

GNNs are particularly interesting because they promise to bring the performance of deep learning models [8] to graph-based domains. In particular, convolutional neural networks (CNNs) are nowadays the de-facto standard for processing image data. CNNs exploit the image structure by performing spatial convolutions (with a limited receptive field) on the image, thus increasing parameter sharing and lowering their complexity. A number of authors recently have explored the possibility of extending CNNs to the graph domain by several generalizations of the convolution operator, a trend which has been generically called 'geometric deep learning' [6]. In one of the first proposals [9], graph Fourier transform was used at every layer of the network to perform filtering operations in a graph-frequency domain. However, this approach was not scalable as it scaled linearly with the size of the graph. Defferard et al. [10] extended this approach by using polynomial filters on the frequency components, which can be rewritten directly in the graph domain, avoiding the use of the graph Fourier transform. A further set of modifications was proposed by Kipf and Welling [11] (described more in depth in Section II), resulting in a generic graph convolutional network (GCN). GCN has been successfully applied to several real-world scenarios, including semi-supervised learning [11], matrix completion [12], program induction [13], modeling relational data [14], and several others.

Like most neural networks, GCNs interleave linear layers, wherein information is adaptively combined according to the topology of the graph, with nonlinear activation functions that are applied element-wise. The information over the nodes can then be combined to obtain a graph-level prediction, or kept separate to obtain a node-specific inference (e.g., predictions on unlabeled nodes from a small set of labeled ones). Most literature for GCNs has worked with very simple choices for the activation functions, such as the rectified linear unit (ReLU) [11]. However, it is well known that the choice of the function can have a large impact on the final performance of the neural network. Particularly, there is a large literature for standard neural networks on the design of flexible schemes for *adapting* the activation functions themselves from the training data [15]–[18]. These techniques range from the use of simple parametrizations over known functions (e.g., the parametric ReLU [15]), to the use of more sophisticated non-parametric families, including the Maxout network [16], the adaptive piecewise linear unit [17], and the recently proposed kernel activation function (KAF) [18].

Contribution of the paper: In this paper, we conjecture that choosing properly the activation function (beyond simple ReLUs) can further improve the performance of GCNs, possibly by a significant margin. To this end, we enhance the basic GCN model by extending KAFs for the activation functions of the filters. Each KAF models a one-dimensional activation function in terms of a simple kernel expansion (see the description in Section III). By properly choosing the elements of this expansion beforehand, we can represent each function with a small set of (linear) mixing coefficients, which are adapted together with the weights of the convolutional layers using standard back-propagation. Apart from flexibility,

ISBN 978-90-827970-1-5 © EURASIP 2018

the resulting scheme has a number of advantages, including smoothness in its domain and possibility of implementing the algorithm using highly vectorized GPU operations. We compare on two benchmarks for semi-supervised learning, showing that the proposed GCN with KAFs can significantly outperform all competing baselines with a marginal increase in computational complexity (which is offset by a faster convergence in terms of epochs).

Structure of the paper: Section II introduces the problem of inference over graphs and the generic GCN model. The proposed extension with KAFs is described in Section III. We evaluate and compare the model in Section IV before concluding in Section V.

II. Graph convolutional networks

A. Problem setup

Consider a generic undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \dots, N\}$ is the set of N vertices, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges connecting the vertices. The graph is equivalently described by the (weighted) adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, where its generic element $a_{ij} \geq 0$ if and only if nodes i and j are connected. A graph signal [1] is a function $f : \mathcal{V} \to \mathbb{R}^F$ associating to vertex n a F-dimensional vector $\mathbf{x}_n$ (each dimension being denoted as a channel). For a subset $\mathcal{L} \subset \mathcal{V}$ of the vertices we have also available a node-specific label y_n, which can be either a real number (graph regression) or a discrete quantity (graph classification). Our task is to find the correct labels for the (unlabeled) nodes that are not contained in $\mathcal{L}$. Note that if the graph is unknown and/or must be inferred from the data, this setup is equivalent to standard semi-supervised (or, more precisely, transductive) learning.

Ignoring for a moment the graph information, we could solve the problem by training a standard (feedforward) NN to predict the label y from the input $\mathbf{x}$ [8]. A NN is composed by stacking L layers, such that the operation of the lth layer can be described by:

$$\mathbf{h}_l = g\left(\mathbf{W}_l \mathbf{h}_{l-1}\right) , \tag{1}$$

where $\mathbf{h}_{l-1} \in \mathbb{R}^{N_{l-1}}$ is the input to the layer, $\mathbf{W}_l \in \mathbb{R}^{N_l \times N_{l-1}}$ are trainable weights (ignoring for simplicity any bias term), and $g(\cdot)$ is a element-wise nonlinear function known as activation function (AF). The NN takes as input $\mathbf{x} = \mathbf{h}_0$, providing $\widehat{\mathbf{y}} = \mathbf{h}_L \in \mathbb{R}^C$ as the final prediction. A number of techniques can be used to include unlabeled information in the training process of NNs, including ladder networks [19], pseudo-labels [20], manifold regularization [2], and neural graph machines [21]. As we stated in the introduction, however, the aim of GNNs is to include the proximity information contained in $\mathbf{A}$ directly *inside* the processing layers of NNs, in order to further improve the performance of such models. As shown in the experimental section, working directly in the graph domain can obtain vastly superior performances as compared to standard semi-supervised techniques.

Note that if the nodes of the graphs are organized in a regular grid, spatial information can be included by adding standard convolutional operations to (1), as is common in CNNs [8]. In order to extend this idea to general unweighted graphs, we need some additional tools from the theory of graph signal processing.

B. Graph Fourier transform and graph neural networks

In order to define a convolutional operation in the graph domain, we can exploit the normalized Laplacian matrix $\mathbf{L} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$, where $\mathbf{D}$ is a diagonal matrix with $D_{nn} = \sum_{t=1}^{N} A_{tn}$. We denote the eigendecomposition of $\mathbf{L}$ as $\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^T$, where $\mathbf{U}$ is a matrix collecting column-wise the eigenvectors of $\mathbf{L}$, and $\mathbf{\Lambda}$ is a diagonal matrix with associated eigenvalues. We further denote by $\mathbf{X} \in \mathbb{R}^{N \times F}$ the matrix collecting the input signal across all nodes, and by $\mathbf{X}_i \in \mathbb{R}^N$ its ith column. The graph Fourier transform of $\mathbf{X}_i$ can be defined as [1]:

$$\hat{\mathbf{X}}_i = \mathbf{U}^T \mathbf{X}_i \, , \tag{2}$$

and, because the eigenvectors form an orthonormal basis, we can also define an inverse Fourier transform as $\mathbf{X}_i = \mathbf{U}\hat{\mathbf{X}}_i$. We can exploit the graph Fourier transform and define a convolutional layer operating on graphs as [9]:

$$\mathbf{H}_1 = g\left(\sum_{i=1}^{F} \mathbf{U} h(\mathbf{\Lambda}; \mathbf{\Theta}_i) \mathbf{U}^T \mathbf{X}_i\right) , \tag{3}$$

where $h(\mathbf{\Lambda}; \mathbf{\Theta}_i)$ is a (channel-wise) filtering operation acting on the eigenvalues, having adaptable parameters $\mathbf{\Theta}_i$. The previous operation can then be iterated to get successive representations $\mathbf{H}_2, \mathbf{H}_3, \dots$ up to the final node-specific predictions. The choice of $h(\cdot)$ determines the complexity of training. In particular, [10] proposed to make the problem tractable by working with polynomial filters over the eigenvalues:

$$h(\mathbf{\Lambda}; \mathbf{\Theta}_i) = \sum_{k=0}^{K-1} \Theta_{ik} T_k(\widetilde{\mathbf{\Lambda}}) \, , \tag{4}$$

where $\widetilde{\mathbf{\Lambda}} = 2\mathbf{\Lambda}/\lambda_{\text{max}} - \mathbf{I}$ (with λ_{max} being the largest eigenvalue of $\mathbf{L}$), and $T_k(\cdot)$ the Chebyshev polynomial of order k defined by the recurrence relation:

$$T_k(\widetilde{\mathbf{\Lambda}}) = 2\widetilde{\mathbf{\Lambda}} T_{k-1}(\widetilde{\mathbf{\Lambda}}) - T_{k-2}(\widetilde{\mathbf{\Lambda}}) \, , \tag{5}$$

with $T_0(\widetilde{\mathbf{\Lambda}}) = 1$ and $T_1(\widetilde{\mathbf{\Lambda}}) = \widetilde{\mathbf{\Lambda}}$. The filtering operation defined by (4) is advantageous because each filter is parameterized with only K values (with K chosen by the user). Additionally, it is easy to show that the filter is localized over the graph, in the sense that the output at a given node depends only on nodes up to maximum K hops from it. Thanks to the choice of a polynomial filter, we can also avoid the expensive multiplications by $\mathbf{U}$ and $\mathbf{U}^T$ by rewriting the filtering operation directly on the original graph domain [10].

In order to avoid the need for the recurrence relation, [11] proposed to further simplify the expression by setting $K = 1$, $\theta_{i0} = -\theta_{i1}$ for all channels, and assuming $\lambda_{\text{max}} = 2$. Back-substituting in the original expression (3) we obtain:

$$\mathbf{H}_1 = g\left(\left(\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\right)\mathbf{X}\mathbf{\Theta}_1\right) , \tag{6}$$

ISBN 978-90-827970-1-5 © EURASIP 2018

where $\mathbf{\Theta}_1 \in \mathbb{R}^{F \times N_1}$ is the matrix of adaptable filter coefficients. Practically, we can further avoid some numerical instabilities (due to the range of the eigenvalues) by substituting $\mathbf{I} + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ with $\widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}$, where $\widetilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\widetilde{D}_{nn} = \sum_{t=1}^{N} \widetilde{A}_{tn}$. The previous expression can then be iterated as in (1) to obtain a graph convolutional network (GCN). As an example, a GCN with two layers is given by:

$$\hat{\mathbf{Y}} = g\left(\widehat{\mathbf{A}} g\left(\widehat{\mathbf{A}}\mathbf{X}\mathbf{\Theta}_1\right)\mathbf{\Theta}_2\right) , \tag{7}$$

where we defined $\widehat{\mathbf{A}} = \widetilde{\mathbf{D}}^{-\frac{1}{2}}\widetilde{\mathbf{A}}\widetilde{\mathbf{D}}^{-\frac{1}{2}}$. The parameters $\{\mathbf{\Theta}_l\}_{l=1}^{L}$ can be initialized randomly and trained by minimizing a suitable loss over the labeled examples, such as the cross-entropy for classification problems [11]. A single layer of the GCN can combine information coming only from the immediate neighbors of a node, while using multiple layers allows information to flow over several hops. In practice, two layers are found to be sufficient for many benchmark problems, and further depth does not provide a benefit [11]. Note, however, that the previous expression requires to choose a proper activation function. This is the subject of the next section.

III. Proposed GCN with kernel activation functions

Up to this point, we assumed that the activation functions $g(\cdot)$ in (7) were given. Note that the functions for different layers (or even for different filters in the same layer) need not be the same. Particularly, the outer function is generally chosen in a task-dependent fashion (e.g., softmax for classification [8]). However, choosing different activation functions for the hidden layers can vastly change the performance of the resulting models and their flexibility.

Most of the research on GNNs and GCNs has focused on simple activation functions, such as the ReLU function:

$$g(s) = \max(0, s) , \tag{8}$$

where s is a generic element on which the function is applied. It is easy to add a small amount of flexibility by introducing a simple parametrization of the function. For example, the parametric ReLU [15] adds an adaptable slope α (independent for every filter) to the negative part of the function:

$$g(s) = \min(0, -\alpha s) + \max(0, s) . \tag{9}$$

As we stated in the introduction, such parametric functions might not provide a significant increase in performance in general, and a lot of literature has been devoted to designing non-parametric activation functions that can adapt to a very large family of shapes. A simple technique would be to project the activation s to a high-dimensional space $\phi(s)$, and then adapt a different linear function on this feature space for every filter. However, this method becomes easily unfeasible for a large dimensionality of $\phi(\cdot)$.

The idea of KAFs [18] is to avoid the expensive feature computation by working instead with a kernel expansion. A known problem of kernel methods is that the elements that we use for computing the kernel values (what is called the dictionary in the kernel filtering literature [22]) can be extremely hard to select. The insight in [18] is to exploit the fact that we are working with one-dimensional functions, by fixing the dictionary beforehand by sampling uniformly D elements (with D chosen by the user) around zero, and only adapting the linear coefficients of the kernel expansion (see the equations below). In this way, the parameter D controls the flexibility of our approach: by increasing it, we increase the flexibility of each filter at the cost of a larger number of parameters per-filter.

More formally, we propose to model each activation function inside the GCN as follows:

$$g(s) = \sum_{i=1}^{D} \alpha_i \kappa(s, d_i) , \tag{10}$$

where d_i are the elements of the dictionary selected according to before, while the mixing coefficients $\{\alpha_i\}_{i=1}^{D}$ are initialized at every filter and adapted independently together with $\{\mathbf{\Theta}_l\}_{l=1}^{L}$. $\kappa(\cdot, \cdot)$ in (10) is a generic kernel function, and we use the 1D Gaussian kernel in our experiments:

$$\kappa(s, d_i) = \exp\left\{-\gamma (s - d_i)^2\right\} , \tag{11}$$

where $\gamma \in \mathbb{R}$ is a free parameter. Since in our context the effect of γ only depends on the sampling of the dictionary, we select it according to the rule-of-thumb proposed in [18]:

$$\gamma = \frac{1}{6\Delta^2} , \tag{12}$$

where Δ is the distance between two elements of the dictionary.

In order to avoid numerical problems during the initial phase of learning, we initialize the mixing coefficients to mimic as close as possible an exponential linear unit (ELU) following the strategy in [18]. Denoting by $\mathbf{t}$ a sampling of the ELU at the same positions as the dictionary, we initialize the coefficients as:

$$\boldsymbol{\alpha} = (\mathbf{K} + \varepsilon\mathbf{I})^{-1}\mathbf{t} , \tag{13}$$

where $\boldsymbol{\alpha}$ is the vector of mixing coefficients, $\mathbf{K} \in \mathbb{R}^{D \times D}$ is the kernel matrix computed between $\mathbf{t}$ and the dictionary, and we add a diagonal term with $\varepsilon > 0$ to avoid degenerate solutions with very large mixing coefficients (see Fig. 1 for an example).

For more details on (10) we refer to the original publication [18]. Here, we briefly comment on some advantages of using this non-parametric formulation. First of all, (10) can be implemented easily in most deep learning libraries using vectorized operations, adding only a marginal overhead (constant on D). Second, the functions defined by (10) are smooth over their entire domain. Finally, the mixing parameters $\{\alpha_i\}_{i=1}^{D}$ can be handled like all other parameters, particularly by applying standard regularization techniques (e.g., ℓ_2 or ℓ_1 regularization).

ISBN 978-90-827970-1-5 © EURASIP 2018

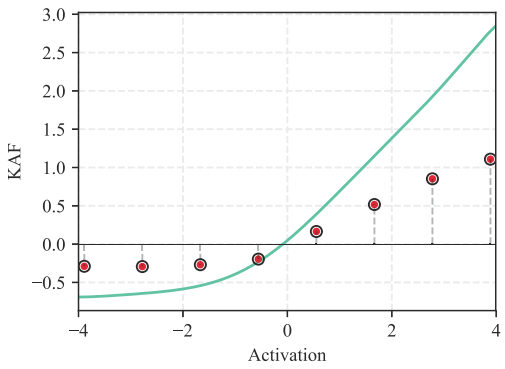

Fig. 1. Example of initializing a KAF according to the ELU function, shown with $D = 8$ and γ chosen according to (12).

IV. Experimental results

A. Experimental setup

We evaluate the proposed model on two semi-supervised learning benchmarks taken from [11], whose characteristics are reported in Table I. Both represents citation networks, where vertices $\mathbf{x}_n$ are documents (each one represented by a bag-of-words describing the relative frequencies of words), edges are links (corresponding to citations across documents), and each document can belong to a given class. The last column in Table I is the percentage of labeled nodes in the datasets.

The baseline network is a two-layer GCN as in (7), while in our proposed method we simply replace the inner activation functions with KAFs, with a dictionary of $D = 20$ elements sampled uniformly in $[-2.0, 2.0]$. We optimize a cross-entropy loss given by:

$$L = \sum_{l \in \mathcal{L}} \sum_{c=1}^{C} Y_{lc} \log \left(\widehat{Y}_{lc} \right) . \tag{14}$$

The loss is optimized using the Adam algorithm [8] in a batch fashion. The labeled dataset is split following the same procedure as [11], where we use 20 nodes per class for training, 1000 elements for validation, and the remaining elements for testing the accuracy. The validation set is used for evaluating the loss at each epoch, and we stop as soon as the loss is not decreasing with respect to the last 10 epochs. All hyper-parameters are selected in accordance with [11] as they were already optimized for the Cora dataset: 16 filters in the hidden layer, an initial learning rate of 0.01, and an additional ℓ_2 regularization on the weights with weighting factor $5e^{-4}$. In addition, we use dropout (i.e., we randomly remove filters from the hidden layer during training) with probability 0.5. The splits for the datasets are taken from [3], and we average over 15 different initializations for the networks.

For the implementation, we extend the original code for the GCN.[1] Specifically, multiplication with the adjacency matrix in (7) is done using efficient sparse data structures, allowing to perform it in linear time with respect to the number of edges. All experiments are run in Python using a Tesla K80 as backend.

[1] https://github.com/tkipf/pygcn

TABLE I
Brief description of the datasets (for more details see the text and [3], [11]).

Dataset	Nodes	Edges	Classes	Features	Labels
Citeseer	3327	4732	6	3703	3.60%
Cora	2708	5429	7	1433	5.20%

B. Experimental results

The results (in terms of accuracy computed on the test set) of GCN and the proposed GCN with KAFs (denoted as KAF-GCN) are given in Table II. We also report the performance of six additional baseline semi-supervised algorithms taken from [3], [11], including manifold regularization (ManiReg), semi-supervised embedding (SemiEmb), label propagation (LP), skip-gram based graph embeddings (DeepWalk), iterative classification (ICA), and Planetoid [11]. References and a full description of all the baselines can be found in the original papers.

Note how, in both cases, the proposed KAF-GCN is able to outperform all the other baselines in a stable fashion (for KAF-GCN, we also report the standard deviation with respect to the different weights' initialization). In particular, results on the Cora dataset represent (to the best of our knowledge) the state-of-the-art results when using such a small labeled dataset. It is important to underline that the gap in performance between GCN and KAF-GCN cannot be reduced by merely increasing the depth (or size) of the former, since its architecture is already fine-tuned to the specific datasets. In particular, [11, Appendix B] reports results showing that neither GCN, nor a variant with residual connections can obtain a higher accuracy when adding more layers. In our opinion, this points to the importance of having adaptable activation functions able to more efficiently process the information coming from the different filters. Due to a lack of space, we are not able to show the shapes of the functions resulting from the optimization process, although they are found to be similar to those obtained by [18], and the interested reader is referred there.

In terms of training time, our implementation of KAF-GCN requires roughly 10% more computation per epoch than the standard GCN for the Cora dataset, and 15% more for the Citeseer one. However, we show in Fig. 2 the (average) loss evolution on the Cora dataset for the two models. We see that KAF-GCN generally converges faster than GCN, possibly due to the higher flexibility allowed by the network. Due to this, KAF-GCN requires a lower number of epochs to achieve convergence (as measured by the early stopping criterion), requiring on average $25/30$ epochs less to converge, which more than compensate for the increased computational time per-epoch.

ISBN 978-90-827970-1-5 © EURASIP 2018

TABLE II
RESULTS IN TERMS OF ACCURACY OVER THE TEST SET. ALL BASELINES ARE TAKEN FROM [3], [11]. THE PROPOSED ALGORITHM IS DENOTED AS KAF-GCN. BEST RESULTS ARE HIGHLIGHTED IN BOLD.

Dataset	ManiReg	SemiEmb	LP	DeepWalk	ICA	Planetoid	GCN	KAF-GCN
Citeseer	60.1	59.6	45.3	43.2	69.1	67.7	70.3	$\mathbf{70.9 \pm 0.1}$
Cora	59.5	29.0	68.0	67.2	75.1	75.7	81.5	$\mathbf{83.0 \pm 0.2}$

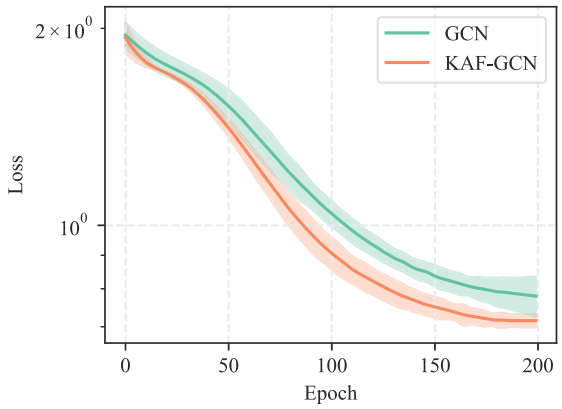

Fig. 2. Loss evolution for GCN and KAF-GCN on the Cora dataset. We show the mean with a solid line and the standard deviation with a lighter color.

V. CONCLUSIONS

In this paper we have shown that the performance of graph convolutional networks can be significantly improved with the use of more flexible activation functions for the hidden layers. Specifically, we evaluated the use of kernel activation functions on two semi-supervised benchmark tasks, resulting in faster convergence and higher classification accuracy.

Two limitations of the current work are the use of undirected graph topologies, and the training in a batch regime. While [11] handles the former case by rewriting a directed graph as an equivalent (undirected) bipartite graph, this operation is computationally expensive. We plan on investigating recent developments on graph Fourier transforms for directed graphs [23] to define a simpler formulation. More in general, we aim to test non-parametric activation functions for other classes of graph neural networks (such as gated graph neural networks), which would allow to process sequences of graphs or multiple graphs at the same time.

ACKNOWLEDGMENT

Simone Scardapane was supported in part by Italian MIUR, GAUChO project, under Grant 2015YPXH4W_004.

REFERENCES

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

[2] Y. Yuan, L. Mou, and X. Lu, "Scene recognition by manifold regularized deep learning architecture," *IEEE Trans. Neural Netw. Learn. Syst*, vol. 26, no. 10, pp. 2222–2233, 2015.

[3] Z. Yang, W. W. Cohen, and R. Salakhutdinov, "Revisiting semi-supervised learning with graph embeddings," in *Proc. 33th International Conference on Machine Learning (ICML)*, 2016.

[4] P. Di Lorenzo, S. Barbarossa, P. Banelli, and S. Sardellitti, "Adaptive least mean squares estimation of graph signals," *IEEE Trans. on Signal and Inf. Process. over Netw.*, vol. 2, no. 4, pp. 555–568, 2016.

[5] D. Romero, M. Ma, and G. B. Giannakis, "Kernel-based reconstruction of graph signals," *IEEE Trans. Signal Process.*, vol. 65, no. 3, pp. 764–778, 2017.

[6] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond Euclidean data," *IEEE Signal Process. Mag.*, vol. 34, no. 4, pp. 18–42, 2017.

[7] F. Gama, G. Leus, A. G. Marques, and A. Ribeiro, "Convolutional neural networks via node-varying graph filters," in *Proc. 2018 IEEE Data Science Workshop (DSW)*, 2018, pp. 1–5.

[8] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016.

[9] J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun, "Spectral networks and locally connected networks on graphs," in *Proc. International Conference on Learning Representations (ICLR)*, 2014.

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

[11] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *Proc. International Conference on Learning Representations (ICLR)*, 2017.

[12] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," *arXiv preprint arXiv:1706.02263*, 2017.

[13] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[14] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," *arXiv preprint arXiv:1703.06103*, 2017.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proc. IEEE Int. Conf. on Comput. Vision (ICCV)*, 2015, pp. 1026–1034.

[16] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," in *Proc. 30th International Conference on Machine Learning (ICML)*, 2013.

[17] F. Agostinelli, M. Hoffman, P. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," *arXiv preprint arXiv:1412.6830*, 2014.

[18] S. Scardapane, S. Van Vaerenbergh, S. Totaro, and A. Uncini, "Kafnets: kernel-based non-parametric activation functions for neural networks," *arXiv preprint arXiv:1707.04035*, 2017.

[19] A. Rasmus, M. Berglund, M. Honkala, H. Valpola, and T. Raiko, "Semi-supervised learning with ladder networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 3546–3554.

[20] D.-H. Lee, "Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks," in *Workshop on Challenges in Representation Learning, ICML*, vol. 3, 2013, p. 2.

[21] T. D. Bui, S. Ravi, and V. Ramavajjala, "Neural graph machines: Learning neural networks using graphs," *arXiv preprint arXiv:1703.04818*, 2017.

[22] W. Liu, J. C. Principe, and S. Haykin, *Kernel adaptive filtering: a comprehensive introduction*. John Wiley & Sons, 2011.

[23] S. Sardellitti, S. Barbarossa, and P. Di Lorenzo, "On the graph Fourier transform for directed graphs," *IEEE J. Sel. Topics Signal Process.*, vol. 11, no. 6, pp. 796–811, 2017.