A survey of Graph Neural Network models and their application in the data-driven approach to computationally hard inverse problems on graphs.

Zsolt Pajor-Gyulai Luca Venturi

May 14, 2018

Abstract

Many important tasks in discrete mathematics on graph domains are defined in terms of computationally hard optimization problems. This means that there are no known algorithms computing the solution with algorithmic complexity growing as a polynomial in the size of the problem. In this note, we focus on the learnability of such algorithms from solved instances of the problem using a data-driven approach. It is natural to assume that the most reasonable neural network models would be able to process graphs directly without any preprocessing. To this end, several different Graph Neural Network (GNN) architectures have been proposed ([4, 25, 44]) recently and our first aim here is to attempt a more or less comprehensive review of the available literature. In the second part of this note, we will give an exposition of several recent contributions, where GNN models have been used to learn algorithms for computationally hard problems on graphs: graph matching, travelling salesman problem [39], community detection [3]. We discuss the results and some open questions.

1 Introduction

In every learning task, the selection of good features to represent the data is of paramount importance. However, automating the feature selection process is a highly non-trivial endeavour and consequently, the traditional approach to most pattern recognition tasks, developed since the 50's, relied heavily on hard coded feature extractors. These manual methods were based on the researcher's subjective intuition and imagination and thus suffered from serious tractability limitations for complicated tasks.

The main novelty ushered in by deep learning is that good features can be learned from the data as well ([32]). Once sufficient data and computers strong enough to process it became available, deep neural network architectures achieved enormous success in a variety of learning tasks, in computer vision, speech recognition, text classification, etc. For a recent comprehensive overview see ([17]).

Convolutional Neural Networks (CNN) are variants that are particularly successful in extracting highly meaningful statistical patterns from large-scale and high dimensional datasets, e.g. images, audio, etc. Through the use of convolutional filters, they exploit the local stationarity property of the input data by revealing local features that are shared across the data domain. This has led to important breakthroughs in image, video, and sound recognition tasks underlying many recent technological advances. The emergence of Recurrent Neural Networks (RNN-s) with Long Short Term Memory (LSTM) led to further improvements in this direction by providing methods particularly suitable for processing sequential data.

Up until recently, however, neural network architectures were mostly developed for data lying on regular natural domains like low dimensional Euclidean grids. On the other hand, many datasets of everyday practical interest lie on irregular or non-Euclidean domains. Graphs, for example, are generic forms of representing data with heterogeneous pairwise relationships, which are thus capable of describing the geometric structure of the data domain. For example, user data on social networks, gene data on biological regulatory networks, transportation networks, log data on telecommunication networks, or text documents on word embeddings can naturally be represented on a graphs.

The traditional machine learning approach for graph structured data is to map the graph signal to a simple, mostly Euclidean representation ([22], also [28], [46]) or to define graph features in terms of random walks on graphs (e.g., [43]). However, important topological information may be lost during this preprocessing step, moreover, the exact nature of this mapping might have a profound (and often unpredictable) effect on the outcome of the learning process. To mitigate this loss of information, several approaches has been proposed to preserve the graph structure as long as possible before the processing phase ([15], [48]). Of course, it is more natural to design models and algorithms that are capable of directly dealing with graph structured data.

It is therefore natural to seek to design neural network architectures that are capable of extracting meaningful features from graph signals, while also incorporating the information encoded by the underlying graph structure. The main obstacle is that classical data processing tools were designed to work on regular data domains, where natural notions of translation, modulation, and resolution are available. In addition, most of the previously mentioned applications involve graphs with a large number of vertices creating the need for efficient signal processing algorithms based on localized operations. Finally, while in some applications (e.g. social networks) the graph structure is given, in many other datasets, the details of the graph structure is not immediately clear and often times has to be discovered from the data itself. To summarize, the main challenges of processing graph signals are ([47]):

1. Discovering the underlying graph structure natural for the data.

2. Incorporating the graph structure into localized transform methods, while leveraging the intuition from our understanding of signals on Euclidean domains.

3. Developing computationally efficient implementations of the localized transforms.

In this work, our first goal is to review the literature on available neural network constructions on graphs in Section 2. Our second goal revolves around an interesting application of GNN-s. It is a natural question to ask whether graph based neural network architectures can contribute towards solving classical graph problems, like the travelling salesman problem, graph matching, community detection, Hamiltonian cycles, coloring problems, etc. from labeled training examples. This approach seems particularly promising when the training examples are cheap to generate, i.e., the forward problem is easy (e.g. planting communities, building a graph with a Hamiltonian cycle etc.), but the inverse problem is hard. We review a couple of recent attempts in this direction in Section 3.

2 Neural networks on graphs

Structured domains are characterized by complex patterns which are usually represented as lists, trees and graphs of variable sizes and complexity. The ability to recognize and classify these patterns is fundamental for several applications. While classical neural networks are able to classify static information or temporal sequences, they do not allow the input structures to have different sizes.

In most machine learning applications, the goal is to learn a function $f : \mathcal{X} \to \mathcal{Y}$, mapping some domain $\mathcal{X}$ of input data to the output space $\mathcal{Y}$ of predictions. When the input data exhibits some natural graph structure, it makes sense to incorporate this prior topological knowledge into the input space. In the most general setting (with the exception of signals only being defined on vertices), an input example is then a triple (G, W, F), where $G = (V, E)$ is a graph, $W \in \mathbb{R}^{|V|\times|V|}$ is a matrix of connection weights (naturally, $W_{vv'} = 0$ when $(v, v') \notin E$), and some d-dimensional signal $F \in \mathbb{R}^{|V|\times d}$ on the vertices of the graph. One can think of the graph signal F as a collection of labels $F_v \in \mathbb{R}^d$ attached to each vertex $v \in V$. The output is usually a vector of reals and thus we set $\mathcal{Y} = \mathbb{R}^m$. The intuitive idea is that vertices represent objects or concepts and edges represent their relationship. Note that the above definition includes both directed and undirected graphs, where the latter types have a symmetric connection weight matrix $W^T = W$, also known as the *similarity matrix*. On the other hand, multiple edges can be encoded in the magnitude of the corresponding weight, when this is appropriate.

For a vertex $v \in V$, we define the in- and out-neighborhood as

$$ne_{in}[v] = \{v' \in V : W_{v'v} > 0\}, \qquad ne_{out}[v] = \{v' \in V : W_{vv'} > 0\},$$

and $ne[v] = ne_{in}[v] \cup ne_{out}[v]$. For undirected graphs, we simply write $ne_{in}[v] = ne_{out}[v] = ne[v]$. Sometimes it is advantageous to differentiate the neighbors of v, which we can achieve using a one-to-one function $\mu_v : ne[n] \to \mathbb{N}$ to assign a different *position* to each neighbor. Such graphs are generally referred to as *positional*.

For example, an image can be represented by a Region Adjacency Graph (RAG), where the nodes denote the homogeneous regions of the image and the edges represent their adjacency relationship. In object localization, the goal is to classify the vertices according to whether the corresponding region belongs to the object or not. In this case, the graph G

is given, and the first layer of the graph signal is given by the indicator function of a node that we are trying to classify, i.e. $F_{v0} = \delta_v$. Further layers of the graph signal might encode properties of the corresponding regions, e.g., area, perimeter, etc. As we are trying to infer some property of a single vertex at a time, this is an example of a *vertex focused* task. Accordingly then the function we seek to learn is customarily denoted as $f(G, v)$.

On the other hand, in image classification where we try to classify an image represented by G depending on the object it depicts, e.g., houses, cars, cats, etc. In this case we do not single out individual vertices and therefore this is a *graph focused* task. Accordingly then the function we seek to learn is customarily denoted as $f(G)$.

We can also classify problems according to the type of information required to compute the values of the prediction function f.

1. *Connection based problems*, where the prediction function depends only on the topological information encoded in the graph-connectivity. For example, we may try to learn a function that identifies cliques or one that finds the number of neighbors of each nodes.

2. *Label based problems*, where the prediction function can be computed from the graph signals, often times using a single one. One such task is given by the case when the graph signal is a Boolean vector on each node, i.e. $F \in \{0, 1\}^{|V| \times d}$ and the function we try to learn is the total number of ones.

3. *General problems*, that use both type of information, for example, learning the total degree function. It turns out that often times it is advantageous to treat a connection based problem as a general one as looking at the labels can help our algorithm distinguish between the nodes (see e.g. [18]).

There are two immediate questions one has to consider when designing neural network architectures capable of incorporating information about the graph structure.

1. How to find graph structure in a particular dataset where it is not naturally given?

2. How to propagate information through the graph to produce the output?

We address the first question in Section 2.1, while we present the two main methods to answer the second one in Section 2.2 and Section 2.3.

2.1 Learning graph structure from data

While some recognition tasks have prior knowledge of the graph structure of the input data, many other real-world applications do not have such a knowledge, in which case it is necessary to estimate the similarity matrix W from the data as well. Following [25], we review two examples of such methods.

2.1.1 Unsupervised Graph Estimation

Given N samples of input data in an M dimensional feature space organized into a single matrix $X \in \mathbb{R}^{N\times M}$, the simplest estimate on the similarity matrix can be based on the distance of the feature vectors. Indeed, we may consider the pairwise distances given by

$$d(i,j) = \|X_i - X_j\|^2,$$

where X_i is the i-th column of X an then build the similarity matrix as different versions of Gaussian diffusion kernels ([2], [53])

$$W_{ij} = e^{-\frac{d(i,j)}{\sigma^2}}, \qquad W_{ij} = e^{-\frac{d(i,j)}{\sigma_i \sigma_j}}, \tag{1}$$

where σ_i is taken to be the distance $d(i, i_k)$ between the feature vector i and its k-th nearest neighbor. The advantage of the latter so called self-tuning diffusion kernel is that the variance is locally adapted around each feature point as opposed to the former version where the variance is shared.

The main drawback of this method is that it is only based on second order correlations between features and as a result it ignores higher order statistics that might be non-negligible in certain contexts. On the other hand, this procedure does not require labeled data.

2.1.2 Supervised Graph Estimation

The unsupervised method described above trades of the need for labels to having to work with an arbitrarily chosen distance and is thus unable to extract statistics from the input signal that might not correspond to the subjectively chosen similarity criteria.

One way to discover the notion of feature similarity most suitable for a particular classification task is to use features learned by the first layer of a fully connected neural network. Indeed, for example, given a training set $X \in \mathbb{R}^{N\times M}$ and labels $y \in \{1, \dots, C\}^N$, we may train a K layer fully connected network with weights $W_1, \dots, W_K$ with some nonlinear activation functions in between. We can then use the weight matrix W_1 learned in the first layer to define feature distance as

$$d_{sup}(i,j) = \|W_{1,i} - W_{1,j}\|^2,$$

where $W_{1,i}$ is the ith column of W_1, which can then be fed into (1). This way, we may use the supervised paradigm to extract through W_1 a collection of linear measurements that best serve the classification task.

2.2 Iterative information diffusion

The first works approaching learning features directly of graph based data were studying a special case called *Recursive Neural Networks* ([48], [15], [20]). These are models whose input domain consists of Directed Acyclic Graphs (DAGs) that estimate the parameters of a function that maps DAGs into a vector of reals. More precisely, the output of a layer

of *recursive neurons* at vertex $v \in V$ depends on the output of all vertices $v' \in V$ such that $(v, v') \in E$. Applied to DAGs, this recursive rule can be unrolled and, after topological ordering, computed efficiently in one forward pass. On the other hand, cyclic or non-directed graphs must undergo a preprocessing phase. To generalize this idea to arbitrary graphs, [18], [44] introduced a framework for undirected graphs.

The main idea is to derive a flat description of the information related to each vertex v. This can be implemented by using a vector of real numbers called a *state* $x_v \in \mathbb{R}^D$ for each vertex v. x_v pools local information in the neighborhood of vertex v and it is reasonable to represent it as the output of a parametric function h_w, the *state transition function* that is of the form

$$x_v = h_w(F_v, x_{ne[v]}, F_{ne[v]}), \tag{2}$$

where $x_{ne[v]}$ and $F_{ne[v]}$ denotes the collection of the corresponding quantities over the neighboring nodes. In positional graphs, (2) provides sufficient flexibility to express different response to data coming from different neighbors. In non-positional graphs, however, this introduces an awkward need to artificially order the neighbors. In this case, it is reasonable to treat all neighbors equally and calculate the state x_v as a sum of contributions from each of v's neighbors, namely

$$x_v = \sum_{v' \in ne[v]} h_{0w}(F_v, x_{v'}, F_{v'}).$$

for some function h_{0w}.

Once the local information at each node has its vectorial representation, we can also assign an output o_v evaluated by another parametric function g_w, the *output function*

$$o_v = g_w(x_v, F_v), \qquad v \in V. \tag{3}$$

Stacking together (2) and (3), we obtain the vectorial system

$$\mathbf{x} = \mathbf{H_w}(\mathbf{x}, \mathbf{F}) \tag{4}$$

$$\mathbf{o} = \mathbf{G_w}(\mathbf{x}, \mathbf{F}). \tag{5}$$

Here, we introduced the convention that quantities defined as stacking together corresponding quantities over the vertices are denoted in boldface. Clearly, (4) is recursive with respect to the state and thus $\mathbf{x}$ is well defined only if there is a unique solution. Finding this unique solution is exactly the mechanism through which local information gets dispersed globally over the graph. Given a graph G, the computation can be visualized by substituting all of the vertices with units that compute the function f_w and are connected according to the graph topology. The obtained network is called the *encoding network*. For a vertex focused task, now we can define the output of the model as $f_w(G, v) = o_v$. To obtain the output for a graph focused task requires a minor modification. In fact, one can simply add a dummy vertex v^* connected to all other vertices (in order to be able to pool information globally) and define the output o_{v^*} at v^*.

A training set for the above model can be defined as the set of m examples

$$\mathcal{L} = \{(G_i, v_i, \mathbf{t_i}) | 1 \leq i \leq m\},$$

where each triple $(G_i, v_i, \mathbf{t_i})$ consists of a graph G_i and one of its vertices v_i and a desired output $\mathbf{t_i}$ defined over a subset of vertices $S \subseteq V$ called *supervised nodes*. The L^2 error of the model over this training set can be defined as

$$e_w = \sum_{i=1}^{m} \sum_{v \in S} (t_{iv} - f_w(G, v))^2.$$

This error then drives a backpropagation procedure that adapts the parameters w such that $f_w(G, v)$ approximates t_{iv}.

Several possible implementations of the state transition function and the output functions have been proposed. Recursive networks and Graph Neural Networks (GNNs) differ in these implementation details and the class of graphs that they can process.

2.2.1 Recursive Neural Networks

Recursive Neural Networks ([48], [15], [20]) are the solution for a special case of the full problem, which proposes to solve the complication posed by finding a unique solution to equation (4) by imposing the following constraints:

- Inputs graphs are directed and acyclic;
- The state x_v of vertex v only depends on its children and thus

$$x_v = \sum_{v' \in ne_{out}[v]} h_{0w}(F_v, x_{v'}, F_{v'});$$

This way the recursion can be simply solved by computing the state of each leaf vertex, while any other vertex x_v can be computed once the state of the children $x_{ne_{out}[v]}$ are available. To achieve complete global pooling of information in graph focused problems, each graph must contain a *supersource* vertex r, from which there is a path to all other nodes (such a node can be always added if not present). As supersources are the only cadidate to collect all the information in the structure, it usually is the only node that has a target assigned to it, i.e. $S = \{r\}$.

2.2.2 Graph Neural Networks

For general graphs with cycles, Graph Neural Networks ([18], [44]) use a message passing scheme to compute the solution of (4) and thus the learning algorithm consists of two phases:

(a) The states $x_v(t)$ are iteratively updated using the rule

$$x_v(t+1) = f_w(F_v, x_{ne[v]}(t), F_{ne[v]})$$

until the resulting dynamics (approximately) reaches a stable fixpoint at step T.

(b) The gradient $\frac{\partial e_w(T)}{\partial w}$ is computed and the weights w are updated according to a gradient descent step.

These two phases are repeated until a given stopping criterion is reached implementing a gradient descent strategy on the error e_w. Of course, the above procedure only works if the iteration in (a) actually reaches a stable fixed point. This can be guaranteed (together with exponentially fast convergence) if the function H_w is a *contraction mapping* uniformly in w, that is, if there exists some $\mu \in [0, 1)$, such that for any $\mathbf{x}_1, \mathbf{x}_2$, we have

$$\|\mathbf{H}_w(\mathbf{x}_1) - \mathbf{H}_w(\mathbf{x}_2)\| \leq \mu \|\mathbf{x}_1 - \mathbf{x}_2\|.$$

There could be many possible ways to implement the two functions h_w and g_w, although it is popular to use a three layer feedforward network due to their universal approximation property. On the other hand, the need to keep $\mathbf{H}_w$ a contraction mapping limits the values that the weights w can assume. One way to achieve this is to recall that if the induced norm of Jacobian matrix $\partial \mathbf{H}_w / \partial \mathbf{x}$ is bounded by a number smaller than one, then H_w is a contraction mapping and then force the weights into sufficiently small values for this condition to be satisfied. This approach, however, leads to the loss of the global approximation property as the weights are not free to vary anymore. An alternative solution is to add a penalty to the error function,

$$e_w = \sum_{i=1}^{p} \sum_{v \in S} (t_{iv} - f_w(G, v))^2 + \beta L\left(\left\| \frac{\partial \mathbf{H}_w}{\partial \mathbf{x}} \right\|_1\right),$$

where the function L is given by

$$L(y) = \begin{cases} (y - \mu)^2 & y > \mu \\ 0 & \text{otherwise} \end{cases}$$

and β is a hyperparameter balancing the relative importance of the error on the supervised output and the deviation from the the desired contraction constant μ. Of course, this does not guarantees that the norm remains always strictly under μ, however, the risk of instability turns out to be very low in practice, as the contraction property is not actually necessary for convergence.

It was proved in [44] that under the contraction mapping assumption, the output function $f_w(G, v)$ is differentiable with respect to the weights, and therefore there is no theoretical argument against gradient based learning. Moreover in the same paper, an efficient learning algorithm was proposed replacing the original Almeida-Pineda algorithm used in [18] by a variant we now describe. Namely, they show that if we define the backward dynamical system z by

$$z(t) = z(t + 1) \frac{\partial \mathbf{H}_w}{\partial \mathbf{x}}(\mathbf{x}, F) + \frac{\partial e_w}{\partial \mathbf{o}} \frac{\partial \mathbf{G}_w}{\partial \mathbf{x}}(x, F)$$

then the sequence $z(T), z(T - 1), \ldots$ converges to a limiting vector z exponentially fast independently of the initial state $z(T)$. Then the gradient of the loss function with respect to the weights can be written as

$$\frac{\partial e_w}{\partial w} = \frac{\partial e_w}{\partial \mathbf{o}} \frac{\partial \mathbf{G}_w}{\partial w}(\mathbf{x}, F) + z \frac{\partial \mathbf{H}_w}{\partial w}(\mathbf{x}, F).$$

The advantage of this algorithm over backpropagation through time, used in similar RNN-like settings, is that after running the propagation to convergence, the gradients are computed solely based upon the converged solution. This way, it is not necessary to store the intermediate states, which leads to considerable savings in terms of memory.

Recursive networks and GNN-s were successfully applied to graph problems including subgraph matching, where the network needs to decide whether a particular graph is contained in the examples as a subgraph and the mutagenesis problem, where the goal is to identify mutagenic compounds based on molecular descriptions. The two types of graph network approaches (in situations where both are available) were compared in [12], and [49] with GNN-s showing better performance. The computational cost aspect of GNNs were analyzed in [44]. For a similar network construction, see also [35].

2.2.3 Gated Graph Neural Networks

There are two main disadvantages of the GNN-model described above: (1) the parameters must be constrained so that the propagation step is a contraction map, possibly limiting the expressivity of the model; (2) the model is not well suited to output sequential data. On the other hand, there is evidence that contraction maps have trouble propagating information across a long range in a graph. Also, many problems with graph inputs require outputting sequences, for example, paths on a graph, enumerations of graph nodes with desirable properties, or sequences of global classifications mixed with a start and an end node.

Combining the ideas from the GNN model with Gated Recurrent Units (GRU)-s, an improved variant of the recurrent unit in RNN, [33] proposed a modification of the GNN scheme called Gated Graph Neural Networks (GG-NN-s). Here the states of the vertices are initialized by padding the vertex signals to size D:

$$x_v^{(1)} = \begin{pmatrix} F_v \\ 0 \end{pmatrix}, \qquad \mathbf{x}^{(1)} = \begin{pmatrix} x_1^{(1)} \\ \vdots \\ x_{|V|}^{(1)} \end{pmatrix},$$

where the latter stacked vector is of size $|V|D$. The communication between the nodes is realized by a propagation matrix $\mathbf{A} \in \mathbb{R}^{kD|V| \times D|V|}$ through the activation vector $\mathbf{a}^{(t)} = \mathbf{A}\mathbf{x}^{(t-1)} + \mathbf{b} \in \mathbb{R}^{kD|V|}$, where $k = 1$ for undirected graphs and $k = 2$ for directed graphs. The matrix $\mathbf{A}$ corresponds to the structure of the edges with a distinction between incoming and outgoing edges in the directed case. The next state is now computed with GRU-like updates first by defining the update and reset gate

$$z_v = \sigma\left(W^z a_v^{(t)} + U^z x_v^{(t-1)}\right), \qquad r_v^{(t)} = \sigma\left(W^r a_v^{(t)} + U^r x_v^{(t-1)}\right),$$

followed by

$$\tilde{x}_v^{(t)} = \tanh\left(W^x a_v^{(t)} + U^x\left(r_v^{(t)} \odot x_v^{(t-1)}\right)\right), \qquad x_v^{(t)} = (1 - z_v^{(t)}) \odot x_v^{(t-1)} + z_v^{(t)} \odot \tilde{x}_v^{(t)}.$$

Here, $W^z, U^z, W^r, U^r, W^x, U^x$ are learnable parameter matrices.

After iterating the above procedure for T steps, one may use several output models based on $\mathbf{x}^{(T)}$ and F. For example, for vertex-focused task, the model

$$o_v = g(x_v^{(T)}, F_v)$$

with some function g was proposed to output vertex scores followed by perhaps a softmax layer to extract classification probabilities. On the other hand, for graph focused tasks, the authors propose using a graph level state

$$x_G = \tanh\left(\sum_{v\in V} \sigma\left(i(x_v^{(T)}, F_v)\right) \odot \tanh\left(j(x_v^{(T)}, F_v)\right)\right),$$

where i and j are some neural networks. The sigmoid part of this expression acts as a soft attention mechanism that decides which nodes are relevant to the current graph-level task.

The architecture described above is suitable for producing a single output. In order to obtain a sequence of outputs, [33] proposes to use several GG-NNs operating iteratively. For each step, there are two GG-NNs $\mathcal{F}_o$ and $\mathcal{F}_F$, the former for predicting the output $\mathbf{o}^{(k)}$, while the latter is used to predict the next graph signal $F^{(k+1)}$. If required, both GG-NNs can have their own propagation and output models, although the former can be shared, which makes the network faster to train and evaluate without much degradation of performance in many cases. As to the output of $\mathcal{F}_F$, the model

$$F_v^{(k+1)} = \sigma\left(j\left(x_v^{k,(T)}, F_v^{(k)}\right)\right)$$

was proposed with some neural network j.

Finally, let us mention that it is possible to treat the intermediate graph signals $F^{(k)}$, $k = 1, \dots T$ as observed and input them directly into the system. This might be desirable where some a priori information is available on what these signals should be.

2.3 Convolutional networks on graphs

A radically different approach to propagating local information through the graph is to generalize the concept of Convolutional Neural Networks (CNNs) on Euclidean domains. As we discussed earlier, the main challenge of this approach is that there is no natural notion of translation on general graphs and thus the classical definition is not immediately applicable.

2.3.1 Convolution on Euclidean grids

Let us start by recalling the convolution operator in its usual setting. CNNs leverage two important ideas that can help improve a machine learning system: *sparse (or finite range) interactions* and *parameter sharing*.

The first idea is closely related to the assumption of *locality* in the data. Assume for example that a data point is a sequence represented as a vector:

$$\mathbf{x} = (x_1, \dots, x_T) \in \mathbb{R}^T \ .$$

Then, loosely speaking, locality in the data means that each component x_t has 'strong' correlation with its 'neighbors' $x_{t-k}, \dots, x_{t+k}$ and 'weak' correlations with components which are 'far apart', $x_1, \dots, x_{t-k}, x_{t+k}, \dots, x_T$. Such locality can be exploited by taking 'sparse' layers. More specifically, if $\mathbf{x}_i = \rho(w_i \mathbf{x}_{i-1})$ is the output of the i-th layer, sparse interactions are obtained by choosing a matrix $w_i \in \mathbb{R}^{d_i \times d_{i-1}}$ such that $w_{i,j,l} = 0$ if $|l - j| > k$. Here d_i is the dimension of the ith layer.

On the other hand, the idea of parameter sharing is closely related to the assumption of *stationarity* in the data. This means, loosely speaking, that the distribution of a chunk of the data $(x_t, \dots, x_{t+k})$ is the same as the distribution of such a chunk translated as $(x_{t+s}, \dots, x_{t+k+s})$. This is exploited in a layer of the network by choosing w_i such that different rows are mere translates of each other.

Exploiting these two properties is then equivalent to take transformations of the type

$$\mathbf{x}_i = \rho\left(\begin{bmatrix} w_{i,1} & \cdots & w_{i,k_i} & 0 & \cdots & 0 \\ 0 & w_{i,1} & \cdots & w_{i,k_i} & \ddots & \vdots \\ \vdots & \ddots & \ddots & & \ddots & 0 \\ 0 & \cdots & 0 & w_{i,1} & \cdots & w_{i,k_i} \end{bmatrix} \mathbf{x}_{i-1}\right) . \tag{6}$$

Here we can introduce the vector $\mathbf{w}_i = (w_{i1}, \dots, w_{ik_i}, 0, \dots, 0) \in \mathbb{R}^{d_{i-1}}$ of the parameters defining the i-th layer usually referred to as a *filter*. Such tranformations are easily generalized to the case where $\mathbf{x}_i$ is a $d_{i-1,1} \times \cdots \times d_{i-1,k}$ tensor of order k and to the case of more filters per layer. In either case, this sparse parameter sharing scheme greatly reduces the number of parameters to be learned. This is usually further enforced by applying pooling and subsampling layers.

The transformation in (6) can be abbreviated as

$$\mathbf{x}_i = \rho\left((\mathbf{w}_i * \mathbf{x}_{i-1})\right) \in \mathbb{R}^{d_i} ,$$

Let us define the vectors

$$f_{jk} = (\mathbf{f}_j)_k = \frac{1}{\sqrt{d_i}} e^{i\frac{2\pi(j-1)(k-1)}{d_i}} , \quad \text{for } j, k \in [d_i] .$$

and the matrix $M = [\mathbf{f}_1 | \dots | \mathbf{f}_{d_i}] \in \mathbb{C}^{n \times n}$. The vectors $\mathbf{f}_1, \dots, \mathbf{f}_{d_i}$ define an orthonormal basis of $\mathbb{C}^{d_i}$, called the (discrete) Fourier basis. The (discrete) Fourier transform of a vector $\mathbf{x} \in \mathbb{R}^{d_i}$ is accordingly defined as

$$\hat{\mathbf{x}} = M^* \mathbf{x} = \begin{bmatrix} \langle \mathbf{x}, \mathbf{f}_1 \rangle \\ \vdots \\ \langle \mathbf{x}, \mathbf{f}_{d_i} \rangle \end{bmatrix} .$$

In particular, it holds that $\mathbf{x} = M\hat{\mathbf{x}} = \sum_{j=1}^{d_i} \langle \mathbf{x}, \mathbf{f}_j \rangle \mathbf{f}_j$ and that

$$M^*(\mathbf{w} * \mathbf{x}) = (M^* \mathbf{w}) \circ (M^* \mathbf{x}) ,$$

for all $\mathbf{w}, \mathbf{x} \in \mathbb{R}^n$, where $\circ$ denotes the Hadamard (elementwise) product. This latter is probably the most important property, it allows us to express $*$ as a pointwise operation

$$\mathbf{w} * \mathbf{x} = M((M^*\mathbf{w}) \circ (M^*\mathbf{x})) \tag{7}$$

giving us the spectral formulation of the covolution. A similar description can be made in the case where $\mathbf{x}$ is a order k tensor.

The question we address in this section is how can we possibly generalize these two definitions (spatial and spectral) to general graph structures, or more precisely, to data instances of the form $F : V \to \mathbb{R}$, where V is the set of vertices of a graph G.

2.3.2 The spatial approach

The most immediate generalisation of CNNs to general graphs is to consider multiscale, hierarchical local receptive fields as described in [4] adapting an idea of [7] and [19] on automating the selection of receptive fields for deep networks.

On the grid, the dyadic clustering behaves nicely with respect to the underlying Euclidean metric and the translation structure captured by the Laplacian operator. On the other hand, while there is a vast literature on forming multiscale clusterings on graphs ([27], [11], [51], [31]), finding those that are guaranteed to behave well with respect to the graph Laplacian is still an open area of research. [4] proposes a naive agglomerative clustering method we now describe. For other possible strategies to perform hierarchical agglomerative clustering, see [16].

Assume that we wish to have K scales for a K layer network on the graph. We define $V^0 = V$, $W^0 = W$, and describe a construction to obtain a graph $G^k = (V_k, E_k)$ and a connection weight matrix W^k for each layer such that the k-th layer clusters V_k are partitions of V_{k-1} into d_k clusters for every $k = 1, \dots K$ according to the connection weights contained in W^{k-1}. In fact, for every $k = 1, \dots, K$ and $v, v' \in V_k$, we recursively define V_k as an ϵ-covering of V_{k-1} with respect to the matrix W^{k-1}, that is as a partition $V_k = \{\mathcal{P}_1, \dots, \mathcal{P}_{d_k}\}$ of V_{k-1}, such that

$$\sup_{i=1,\dots,d_k} \sup_{v,v' \in \mathcal{P}_i} W^k_{v,v'} > \varepsilon$$

for some $\varepsilon > 0$. This covering can be obtained by, e.g., a simple greedy selection algorithm. The connection weights between $v, v' \in V_k$ as

$$W^k(v, v') = \text{rownormalize}\left(\sum_{s \in v} \sum_{t \in v'} W^{k-1}_{st}\right),$$

where we recall that $v \in V_k$ is by definition a set of vertices in V_{k-1}.

Recall that the goal is to define the analogue of the convolution operator with some sparse filters to get locally connected networks. The receptive fields of these filters are defined using the notion of a neighborhood based on the connection weight matrix through, e.g., a threshold $\delta > 0$ as

$$\mathcal{N}^k_\delta(v) = \{v\} \cup \{v' \in V : W^k_{vv'} > \delta\}, \qquad v \in V_k. \tag{8}$$

This way, we can consider a collection of neighborhoods around each element of V_{k-1}:

$$\mathcal{N}^k = \{\mathcal{N}_{k,v} : v \in V_{k-1}\}, \tag{9}$$

where the dependence on the choice of δ is suppressed. (9) can also be written as $\mathcal{N}_k = \text{supp}_\delta(W^k)$, where supp_δ denotes the row-wise operator selecting the vertices that survive the thresholding in (8). Then, if $\mathbf{x}_k \in \mathbb{R}^{d_{k-1} \times f_{k-1}}$ is the input to layer k, then its output is defined as

$$x_{k+1,v} = L_k h\left(\sum_{v' \in V_k} w_{k,v,v'} x_{k,v'}\right),$$

where $w_{k,v,v'} \in \mathbb{R}^{d_{k-1} \times d_{k-1}}$ are sparse matrices with nonzero entries only in the locations given by $\mathcal{N}^k$, and L_k is a pooling operation over each cluster in V_k. Clearly, $\mathbf{x}_0 = F$, $d_0 = |V|$, and $f_0 = d$. Thus each layer of the network transforms an f_{k-1} dimensional graph signal on V_k into an f_k dimensional signal on V_k, trading-off spatial resolution with the newly created feature coordinates.

This construction reduces the number of learnable parameters from $\mathcal{O}(|V_k||V_{k+1}|f_{k-1}f_k)$ for a fully connected layer to $\mathcal{O}(S_k|V_k|f_{k-1}f_k)$, where S_k is the average size of a neighborhood in $\mathcal{N}^k$. In practice, S_k is typically somewhere between 1 and 4, making this reduction of parameters significant.

A similar spatial generalizations of CNNs to 3D meshes was proposed in [34], while another variant capable of handling changing graph structure in the input examples was introduced in [13], [38]. The resulting architecture was successfully used to learn molecular fingerprints in [13].

2.3.3 The spectral approach

In this section, we are interested in generalizing the spectral definition of the convolution to an equivalent tranformation for signals defined on graphs. More specifically, we consider a connected graph $G = (V, E, W)$ with a signal $F : V \to \mathbb{R}^d$. There are two possible ways to define Fourier transform and through that convolutions on a graph:(1) through the graph Laplacian; (2) through the adjacency matrix.

Convolution on graphs through the graph Laplacian Throughout this section we assume that the graph G is undirected, meaning that the weight matrix W is symmetric. The key to define Fourier transform on graph is to generalize the classical connection between the Laplacian on Euclidean grids/spaces. Therefore in spectral graph analysis, a pivotal role is played by the graph Laplacian through which the notion of frequency and smoothness can be connected. It appears frequently in two different forms: the combinatorial Laplacian is defined as $L = D - W \in \mathbb{R}^{|V| \times |V|}$ where $D \in \mathbb{R}^{|V| \times |V|}$ is the diagonal degree matrix with $D_{ii} = \sum_j W_{ij}$, while the normalized Laplacian is given by $\mathcal{L} = D^{-1/2} L D^{-1/2} = I - D^{-1/2} W D^{-1/2}$. Since L is a real symmetric positive semidefinite matrix, it has a complete set of orthonormal eigenvectors $\{\mathbf{u}_i\}_{i=0}^{|V|} \subset \mathbb{R}^{|V|}$, known as the graph Fourier basis, and their

associated real non-negative eigenvalues $\{\lambda_i\}_{i=0}^{n-1}$ indexed in decreasing order. These can be identified as the natural frequencies of the graph. Note that $\mathcal{L}$ posesses the same eigenvalues with corresponding eigenvectors $D^{1/2}\mathbf{u}_i$.

The Laplacian can thus be written as $L = U\Lambda U^T$, where $U = [\mathbf{u}_0|\cdots|\mathbf{u}_{|V|-1}] \in \mathbb{R}^{|V|\times|V|}$ and $\Lambda = \mathrm{diag}(\lambda_0, \ldots, \lambda_{n-1}) \in \mathbb{R}^{|V|\times|V|}$. The graph Fourier transform is then defined as

$$\hat{\mathbf{x}} = U^T\mathbf{x} \ , \tag{10}$$

alongside with its inverse $\mathbf{x} = U\hat{\mathbf{x}}$. Proceeding by analogy with the formula (7), one can define the convolutions between two real valued signals $\mathbf{x}, \mathbf{y}$ on G:

$$\mathbf{x} *_G \mathbf{y} = U\big(\big(U^T\mathbf{x}\big) \circ \big(U^T\mathbf{y}\big)\big) \ . \tag{11}$$

It follows that a signal $\mathbf{x}$ is filtered by a filter $\mathbf{w}$ as

$$\mathbf{w} *_G \mathbf{x} = U\mathrm{diag}\big(U^T\mathbf{w}\big)U^T\mathbf{x} = U\Lambda_{\mathbf{w}}U^T\mathbf{x} = L_{\mathbf{w}}\mathbf{x} \ .$$

A generic filter is therefore defined by a diagonal matrix $\Lambda_{\mathbf{w}} \in \mathbb{R}^{|V|\times|V|}$. This is the construction proposed by [4]. There are however three problems: (1) these filters might not be localized in space; (2) the multiplication by U and U^T is expensive; (3) The learning complexity of a filter is $\mathcal{O}(|V|)$.

For the first problem, [4] suggests applying a smoothing kernel, exploiting the analogy that smoothness in Fourier space corresponds to spatial locality. This, however, still does not provide precise control over the local support of the filter. [9] instead proposed to use polynomial filters

$$\Lambda_{\mathbf{w}} = \sum_{j=0}^{k-1} w_j\Lambda^j \ ,$$

where the parameter $\mathbf{w} \in \mathbb{R}^k$ is a vector of polynomial coefficients and Λ is the diagonalized Laplacian in the Fourier basis. The filter operation is therefore defined by the matrix

$$L_{\mathbf{w}} = \sum_{j=0}^{k-1} w_jL^j \ .$$

The locality of the filter is explained by the following fact ([21]):

Fact 1. *For every pair of vertices i, j on G such that $d_G(i, j) > k$ it holds*

$$\big(L^k\big)_{i,j} = 0 \ ,$$

where d_G is the shortest path distance, i.e. the minimum number of edges connecting two vertices on the graph.

Consequently, spectral filters represented by k-th order polynomials of the Laplacian are exactly k-localized. Besides, their learning complexity is $O(k)$, the support size of the filter, and thus the same complexity as classical CNNs. Nevertheless, the cost to filter a signal $\mathbf{x}$ as $\mathbf{z} = L_{\mathbf{w}}\mathbf{x}$ is still high with $O(|V|^2)$ operations because of the multiplication with the Fourier basis U. A possible solution to this problem is to parametrize $L_{\mathbf{w}}$ as a polynomial function which can be computed recursively from L, as k multiplications by a sparse matrix cost $O(k|E|) \ll O(|V|^2)$. One such basis is given by Chebychev polynomials (recursively defined as $T_0(x) = 1$, $T_1(x) = x$ and $T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$ for $k \geq 2$), which form an orthogonal basis of $L^2([-1, 1], dy/\sqrt{1-y^2})$. A ($k$-localized) filter can thus be defined as the truncated expansion

$$L_{\mathbf{w}} = \sum_{j=0}^{k-1} w_j T_j(\tilde{L})$$

where $T_j(\tilde{L}) \in \mathbb{R}^{|V|\times|V|}$ is the Chebyshev polynomial of order k evaluated at $\tilde{L} = 2L/\lambda_{max} - I$, which is the rescaled Laplacian (since Chebyshev polynomials are defined on the domain $[-1, 1]$). Denoting $\tilde{\mathbf{x}}_k = T_k(\tilde{L})\mathbf{x} \in \mathbb{R}^{|V|}$, we can use the recurrence relation to compute

$$\tilde{\mathbf{x}}_k = 2\tilde{L}\tilde{\mathbf{x}}_{k-1} - \tilde{\mathbf{x}}_{k-2}, \qquad \tilde{\mathbf{x}}_0 = \mathbf{x}, \qquad \tilde{\mathbf{x}}_1 = \tilde{L}\mathbf{x}.$$

The entire filtering operation $\mathbf{z} = L_{\mathbf{w}}\mathbf{x} = [\tilde{\mathbf{x}}_0|\cdots|\tilde{\mathbf{x}}_{k-1}]\mathbf{w}$ then achives the desired cost of $O(k|E|)$ operations.

Remark 1. *While the above operations are defined in a strict analogy with the standard discrete case, one choice is debatable. The standard discrete convolution on $\mathbb{R}^{|V|}$ can be recovered by considering the graph given by*

$$V = \{0, \ldots, |V| - 1\}, \qquad E = \{(i + 1, i), (i, i + 1) : i \in V \bmod |V|\}.$$

Nevertheless, in this case, the Fourier basis is chosen to be the basis that diagonalizes the graph Laplacian on $\mathbb{C}^n$ (by exploiting its circulant nature) rather than the basis that diagonalizes it on $\mathbb{R}^n$ (by exploiting its symmetric nature). The localized filtering operations differ based on the choices of the basis.

Convolution on graphs through the adjacency matrix The standard discrete convolution operator between signals on $\mathbb{R}^n$ can also be understood in term of z-transforms. The z-transform of a signal $\mathbf{x} = (x_0, \ldots, x_{n-1}) \in \mathbb{R}^n$ is given by the representation

$$\mathbf{x}(z) = \sum_{i=0}^{n-1} x_i z^{-i}$$

in terms of a formal variable z, called the shift (or delay). The z-transform $\mathbf{x}(z)$ provides a (formal) polynomial representation of the signal that is useful in studying how signals are processed by filters. A filtering operation is defined in terms of z-transforms as

$$\mathbf{z}(z) = \mathbf{w}(z) \cdot \mathbf{x}(z) = \left(\sum_{i=0}^{n-1} w_i z^{-i}\right)\left(\sum_{i=0}^{n-1} x_i z^{-i}\right)$$

where the above denotes a product of polynomials, with elements of order $i \leq -n$ considered modulo n (i.e. $z^{-i+kn} = z^{-i}$ for $i \in 0, \ldots, n-1$ and $k \in \mathbb{Z}$). Such operation corresponds to the discrete convolution $\mathbf{z} = \mathbf{w} * \mathbf{x}$. A particular filter is the *shift* (or *delay*) filter

$$\mathbf{w}_{\text{shift}}(z) = z^{-1}.$$

Applied to a signal $\mathbf{x} = (x_0, \ldots, x_{n-1})$ the shift filter gives the output

$$\mathbf{w}_{\text{shift}} * \mathbf{x} = (x_{n-1}, x_0, \ldots, x_{n-2})$$

An important property in such a framework is *shift invariance*:

$$\mathbf{w}_{\text{shift}}(z) \cdot \mathbf{x}(z) = \mathbf{x}(z) \cdot \mathbf{w}_{\text{shift}}(z).$$

The shift operation can be written in matrix form as

$$\mathbf{w}_{\text{shift}} * \mathbf{x} = A_{\text{shift}} \cdot \mathbf{x}$$

given by the shift matrix

$$A_{\text{shift}} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix} .$$

The matrix A_{shift} can be interpreted as the adjacency matrix of a directed graph G with vertices $V = \{0, \ldots, n-1\}$ and edges $E = \{(i, i+1) : i \in V \bmod n\}$. The i-th row of A_{shift} represents the set of in-edges of node i in G: if there is an entry 1 at column j, then there is an edge from j to i. This correspond to assign a natural sequential structure to data $\mathbf{x} \in \mathbb{R}^n$.

This is the starting point of a possible alternative definition of a convolution operation $*_G$ on a generic graph G in ([40]). A (directed) graph G with n vertices is implicitly defined by a (non-symmetric) adjacency matrix $A \in \{0,1\}^{|V| \times |V|}$. The matrix A can be used to define the shift operation on the graph G. A filter $\mathbf{w}$ is defined corresponding to a shift invariant matrix $H_{\mathbf{w}}$, i.e. one which verifies

$$A \cdot H_{\mathbf{w}} = H_{\mathbf{w}} \cdot A \ .$$

Fact 2. *If the characteristic polynomial $p_A(z)$ and the minimal polynomial $m_A(z)$ of A are equal, then every filter matrix $H_{\mathbf{w}}$ commuting with A is a polynomial in A, i.e.*

$$H_{\mathbf{w}} = h_{\mathbf{w}}(A)$$

with $\text{degree}(h_{\mathbf{w}}(z)) \leq n-1$.

In the case when A admits $|V|$ distinct eigenvalues, we can write

$$A = V\Lambda V^{-1}$$

where $V = [\mathbf{v}_0|\cdots|\mathbf{v}_{n-1}]$ is the matrix of n eigenvectors of A, $\Lambda = \mathrm{diag}(\lambda_0, \ldots, \lambda_{n-1})$ is the diagonal matrix of distinct eigenvalues of A. Then the filter $\mathbf{w}$ is defined by

$$H_{\mathbf{w}} = h_{\mathbf{w}}(A) = V h_{\mathbf{w}}(\Lambda) V^{-1}$$

where $h_{\mathbf{w}}(\Lambda)$ is the diagonal matrix $h_{\mathbf{w}}(\Lambda) = \mathrm{diag}(h_{\mathbf{w}}(\lambda_0), \ldots, h_{\mathbf{w}}(\lambda_{n-1}))$. The filtering operation is therefore defined by

$$\mathbf{z} = \mathbf{w} * \mathbf{x} = V h_{\mathbf{w}}(\Lambda) V^{-1}\mathbf{x} \ ;$$

a filter is k-localized if $\mathrm{degree}(h_{\mathbf{w}}) \leq k - 1$. Such an approach permits, in particular, to recover the classical discrete convolution (as well as the discrete Fourier transform) on signals $\mathbf{x} \in \mathbb{R}^n$. It also has a more intuitive interpretation: a localized filter $\mathbf{w} = (w_0, \ldots, w_{k-1}) \in \mathbb{R}^k$ applied to a signal $\mathbf{x} \in \mathbb{R}^{|V|}$ gives

$$\mathbf{z} = \mathbf{w} *_G \mathbf{x} = \sum_{i=0}^{k-1} w_i A^i \mathbf{x}$$

Therefore, such a filter filters information on k distant neighbors on the graph.

Pooling The second type of standard elements of CNN architectures are pooling layers, which requires meaningful neighborhoods on graphs, where similar vertices are clustered together. This comes as a byproduct in the naive spatial construction described above. On the other hand, it is a priori unclear how to do this in a meaningful way that extends to the convolutional construction as well. The goal is to somehow obtain a multi-scale clustering of the graph that preserves local geometric structures, and at the same time produces a coarser graph at each level, which corresponds to the data domain seen at different resolutions.

This task is known to be NP-hard ([5]) requiring approximations. [9] proposes to use the coarsening phase of the Graculus multilevel clustering algorithm ([11] to minimize the normalized cut objective function. This is a greedy algorithm that, at each coarsening level, picks an unmarked vertex v and matches it with one of its unmarked neighbor v' that maximizes the local normalized cut $W_{vv'}(1/d_v + 1/d_{v'})$. The two matched vertices are then marked and the coarsened weights are set as the sum of their weights. This procedure is repeated until all nodes have been explored.

Pooling operations are carried out many times and must therefore be efficient. [9] proposes a rearrangement of the vertices that makes the graph pooling operation as efficient as regular one dimensional pooling.

2.3.4 Extension to sequential data

As it was the case with the first iterative architectures, convolutional CNN-s described above might not be suitable for sequential input and/or output. [45] proposed two ways of comibing graph convolutional GNN-s and recurrent networks: (1) Stack a convGNN for feature extraction and an LSTM. This is fairly straightforward, however the architecture is not fully on the graph anymore. (2) Take an ordinary LSTM but replace the multiplications by dense matrices with a convolutional kernel. In the Euclidean case, this leads to the convLSTM architecture ([52]), which can be easily generalized to the graph context by using graph convolutions described above. Of course, both of these approaches can be extended to any kind of recursive network.

3 GNNs for computationally hard inverse problems

Many tasks in discrete mathematics belong to the group of computationally hard inverse problems. The known algorithms for such problems either scale exponentially in the size of the problem or are based on some not-verifiable heuristics. Instead of investigating the existence of polynomial-time algorithms for the given problem, one may consider a data-driven approach to learn algorithms from solved instances of the problem. The hope that this is indeed possible is that computational hardness usually focuses on worst case complexities, while real world data is often times not adversarial.

The inverse problems we are interested in can be written as discrete optimization problems: given a K-tuple of graphs $(G_1, \dots, G_K)$ with N vertices, find a function $\mathbf{s} : \prod_{k=1}^{K} V_k \to \mathcal{C}$ (where $\mathcal{C}$ is a discrete set of size C) which achieves

$$\min_{\mathbf{s} \,:\, *} \; h(\mathbf{s}, (G_1, \dots, G_k)) \tag{12}$$

where $*$ denotes some additional constraints on $\mathbf{s}$ and h is some cost function specifying the problem. Given the underlying graph structure of the problem, a possible approach is to learn a GNN that takes as input a K-tuple of graphs $(G_1, \dots, G_K)$ and outputs an approximated value $\mathbf{f}$ of the ground truth function $\mathbf{s}$. In the following, we review the works [3, 39], where the authors describe a GCNN architecture to tackle some computationally-hard inverse problems of the form (12).

3.1 Some computationally hard inverse problems

3.1.1 Community detection

In the community detection problem, a graph $G = (V, E)$ models a population split between C communities. Each vertex $i \in \{1, \dots, N = |V|\}$ of the graph belongs to some community $s(i) \in \mathcal{C} = \{1, \dots, C\}$. The adjacency matrix A of the graph is assumed to resemble the community clustering of the nodes: two nodes i, j with $s(i) = s(j)$ are more likely to be connected ($A_{i,j} = 1$) than two nodes i, j with $s(i) \neq s(j)$. The goal of community detection

is thus to recover the community clustering function $s : V \to \{1, \dots, C\}$ given the adjacency matrix A. In the case of $C = 2$ communities, this problem can be formalized as a *min-cut* problem over the graph G: find

$$\mathbf{s} \in \underset{\substack{s(i)=\pm 1 \\ \mathbf{1}^T\mathbf{s}=0}}{\arg\min} \sum_{i,j=1}^{N} (1 - s(i)s(j))A_{i,j} \ . \tag{13}$$

The constraint $\mathbf{1}^T s = 0$ requires the two communities to have same size ($N/2$) and it avoids trivial solutions; other conditions may be imposed. In particular, if $D = \mathrm{diag}(A\mathbf{1})$ is the degree matrix, and $L = D - A$ is the (un-normalized graph Laplacian), problem (13) can be reformulated as finding

$$\mathbf{s} \in \underset{\substack{s(i)=\pm 1 \\ \mathbf{1}^T\mathbf{s}=0}}{\arg\min} \ \mathbf{s}^T L \mathbf{s} \ .$$

This problem is strictly related with finding the Fiedler vector (the eigenvector associated with the second smallest eigenvalue) of the Laplacian [37]. Problem (13) can be generalized to the case of C communities as finding $s : V \to \mathcal{C}$ such that

$$\mathbf{s} \in \underset{\substack{\mathbf{s}\in\mathcal{C}^N \\ N_c(\mathbf{s})=N_c, \ c=1,\dots,C}}{\arg\min} \sum_{i,j=1}^{N} \mathbb{1}\{s(i) = s(j)\}A_{i,j} \ ,$$

where $N_c(\mathbf{s}) = \sum_{i=1}^{N} \mathbb{1}\{s(i) = c\}$ and $(N_1, \dots, N_C)$ are given community sizes. This problem has been widely studied from the statistical point of view (see, for example, Chapter 9 of [1]) and algorithms based on Semi-Definite Programming have been shown to recover an exact solution in polynomial time, under some statistical assumptions on the model.

3.1.2 Graph matching

In the graph matching problem, one is given a pair of graphs $G_i = (V_i, E_i)$, $i = 1, 2$, with adjacency matrices A_1, A_2. The aim is to find a permutation matrix

$$\Sigma \in \Pi \doteq \left\{ \sum_{i=1}^{N} \mathbf{e}_i \mathbf{e}_{\sigma(i)}^T \in \mathbb{R}^{N\times N} : \sigma \in S_N \right\},$$

where S_N denotes the group of permutations on $\{1, \dots, N\}$, which achieves

$$\max_{\Sigma\in\Pi} \ \mathrm{tr}(A_1 \Sigma A_2 \Sigma) \ . \tag{14}$$

The permutation matrix Σ corresponds to a matching $\sigma : V_1 \to V_2$ we wish to recover. It is easy to see that (14) is equivalent to minimizing $\|A_1\Sigma - \Sigma A_2\|_F$, where $\|.\|_F$ is the Frobenius norm. Note that this minimum is zero if and only if A_1 and A_2 are isomorphic.

Problems of the form (14) are known as Quadratic Assignment Problems (QAPs). The quadratic assignment problem is known to be NP-hard and also hard to approximate [41]. Several methods and heuristics had been proposed to address the QAP; see [14] for a recent review of different methods and numerical comparison. The experiments run in [39] compare SDP from [42] and the LowRankAlign method from [14].

3.1.3 Travelling salesman problem

The travelling salesman problem admits a formulation close to (14). One is given a graph $G = (V, E)$ with adjacency matrix A and a weight matrix W on the edges of the graph. The aim is to find a permutation $\sigma \in S_N$ matrix which achieves

$$\min_{\Sigma \in \Pi} \; W_{\sigma(N),\sigma(1)} + \sum_{i=1}^{N-1} W_{\sigma(i),\sigma(i+1)} \; . \tag{15}$$

The sought permutation σ correspond to a cycle on the vertices V whose cost is minimum. Problem (15) can also be reformulated as a QSP: find

$$\Sigma \in \arg\min_{\Sigma \in \Pi} \; \operatorname{tr}(A \Sigma A_C \Sigma) \; ,$$

where $A_C = \mathbf{e}_N \mathbf{e}_1^T + \sum_{i=1}^{N-1} \mathbf{e}_i \mathbf{e}_{i+1}^T$ is the adjacency matrix of the cycle graph on N nodes. On top of the algorithms addressing the above QAP, other algorithms are available to approximately solve the TSP, some of which are based on some heuristics, such as the Lin-Kernighan TSP Heuristic (see [24] for an efficient implementation).

3.2 Proposed network architectures

As previously explained, our aim is to approximate the sought solution $\mathbf{s}$ to (12) with a GNN $\mathbf{f}$. In the works [3, 39], the authors proposed a GCNN architeture for $\mathbf{f}$. The function $\mathbf{f}$ takes a K-tuple $(G_1, \ldots, G_K)$ and a signal $\mathbf{x} \in \mathbb{R}^{N \times K}$ of signals the vertices of the graph and outputs an approximation $\mathbf{f} \simeq \mathbf{s}$. The proposed architectures involve sharing a K layers GCNN architecture and mainly differ in the choice of the output layer and loss function.

3.2.1 GCNN architecture

The base network $\mathbf{f}$ considered takes as input a graph $G = (V, E)$ and a signal $\mathbf{x} \in \mathbb{R}^N$ on the vertices of the graph, and outputs a signal $\mathbf{y} = \mathbf{f}(\mathbf{x}, G; \Theta) \in \mathbb{R}^{N \times C}$. The numbers of vertices N of the graph is fixed. The network is composed by a sequence of residual convolutional

layers and a final output layer (to be specified):

$$
\begin{aligned}
\text{input:} \qquad & \mathbf{x}^{(0)} = \mathbf{x} \in \mathbb{R}^N \ , \\
k\text{-th layer:} \qquad & \mathbf{x}^{(k)} \in \mathbb{R}^{N \times d} \ , \\
& x^{(k)}_{i,m} = \rho\Bigg(\theta^{(k)}_{1,m} \cdot \mathbf{x}^{(k-1)}_i + \theta^{(k)}_{2,m} \cdot \left(D\mathbf{x}^{(k-1)}\right)_i + \theta^{(k)}_{3,m} \cdot \left(U\mathbf{x}^{(k-1)}\right)_i \\
& \qquad + \sum_{j=0}^{K-1} \theta^{(k)}_{4+j,m} \cdot \left(A^{2^j}\mathbf{x}^{(k-1)}\right)_i \Bigg), \quad \text{for } m = 1, \dots, d/2, \\
& x^{(k)}_{i,m} = \theta^{(k)}_{1,m} \cdot \mathbf{x}^{(k-1)}_i + \theta^{(k)}_{2,m} \cdot \left(D\mathbf{x}^{(k-1)}\right)_i + \theta^{(k)}_{3,m} \cdot \left(U\mathbf{x}^{(k-1)}\right)_i \\
& \qquad + \sum_{j=0}^{K-1} \theta^{(k)}_{4+j,m} \cdot \left(A^{2^j}\mathbf{x}^{(k-1)}\right)_i , \quad \text{for } m = d/2+1, \dots, d, \\
\text{output:} \qquad & \mathbf{y} = \mathbf{x}^{(K+1)} = \mathbf{o}\big(\mathbf{x}^{(K)}; \Theta^{(K+1)}\big) \in \mathbb{R}^{N \times C} \ .
\end{aligned}
$$

Each of the K internal layers, is parametrized by the filters $\Theta^{(k)} = \{\theta^{(k)}_1, \dots, \theta^{(k)}_{K+3}\}$, for $k = 1, \dots, K$, with $\theta^{(k)}_i \in \mathbb{R}^{d \times d}$ for $k > 1$ and $\theta^{(1)}_i \in \mathbb{R}^d$. The output layer depends on the specific model and may also deepend on a parameter $\Theta^{(K+1)}$. The network is therefore defined by the collection of parameters $\Theta = \{\Theta^{(k)}\}_{k=1}^{K+1}$.

The graph convolutions at each internal layer are defined by the matrices $A_j = \min\{1, A^{2^j}\}$, where A is the adjacency matrix of the graph G, following the definition of adjacency-based graph convolution given in Section 2.3. The choice of powers 2^j encodes filters localized on 2^j-hop neighborhoods of each node, and allows to aggregate local information at different scales, while at the same time growing the receptive field rapidly with the depth of the network. This model also includes the degree operator $D = \mathrm{diag}(A\mathbf{1})$ and the average operator $U = N^{-1}\mathbf{1}\mathbf{1}^T$. The former allows the network to recover degree information at each layer, while the latter allows it to broadcast information globally. ReLU activations $\rho(z) = \max\{0, z\}$ are used at each internal layer, alongside with linear residual connections [23], which are put in place to both ease with the optimization when using large number of layers, but also to give the model the ability to perform power iterations. To overcome numerical instabilities, spatial batch normalization [26] is also applied at each internal layer. The network depth K is chosen to be of the order of the graph diameter, so that all nodes obtain information from the entire graph. In sparse graphs with small diameter, this architecture offers excellent scalability and computational complexity.

3.2.2 Output layer and model evalutation

Given a set of solved instances of the considered problem $\mathcal{S} = \{(\mathbf{G}_i = (G_{i,1}, \dots, G_{i,K}), \mathbf{s}_i)\}_{i=1}^m$, where each $\mathbf{G}_i$ is a K-tuple of graphs and $\mathbf{s}_i$ is the corresponding solution to (12), the loss

of the model $\mathbf{f}(\cdot;\Theta)$ is defined as

$$L(\Theta;\mathcal{S}) = \frac{1}{m}\sum_{i=1}^{m} \ell(\mathbf{f}(\mathbf{G}_i;\Theta);\mathbf{s}_i) \ ,$$

where $\ell(\cdot;\cdot)$ is a designed loss depending on the considered problem.

Community detection For this problem we assume that the available dataset consists of a sequence $\{(G_i,\mathbf{s}_i)\}_{i=1}^m$ of graphs $G_i=(V_i,E_i)$ and community clustering $\mathbf{s}_i \in \mathcal{C}^N$, with $\mathcal{C}=\{1,\dots,C\}$ the set of communities. Given a graph G, the network output $\mathbf{f}(G;\Theta)\in\mathbb{R}^{N\times C}$ models the probabilities

$$f(G;\Theta)_{i,c} = P\{s(i)=c\}, \quad \text{for } i\in V,\ c\in\mathcal{C}.$$

The network $\mathbf{f}$ is specified by the architecture in Section 3.2.1, with output layer given by

$$\mathbf{y} = \mathbf{o}\big(\mathbf{x}^{(K)};\Theta^{(K+1)}\big) = \mathrm{softmax}\big(\mathbf{x}^{(K)}\Theta^{(K+1)}\big) \in \mathbb{R}^{N\times C} \ ,$$

or

$$y_{i,c} = \frac{\exp\Big(\theta_c^{(K+1)}\cdot x_i^{(K)}\Big)}{\sum_{c'=1}^{C}\exp\Big(\theta_{c'}^{(K+1)}\cdot x_i^{(K)}\Big)} \ , \quad \text{for } c=1,\dots,C.$$

The output layer depends on a softmax filter $\Theta^{(K+1)} = [\theta_1^{(K+1)}|\cdots|\theta_C^{(K+1)}] \in \mathbb{R}^{d\times C}$; the softmax function is applied row-wise. The input signal is taken to be the degree vector $\mathbf{x}^{(0)} = A\mathbf{1}$. Such a model is then evaluated with the following cross-entropy loss:

$$\ell(\mathbf{f}(G;\Theta);\mathbf{s}) = \inf_{\sigma\in S_C} -\sum_{i=1}^{N}\log f(G;\Theta)_{i,\sigma(s(i))} \ . \tag{16}$$

The infimum in the above loss is over all permutation on the communities set $\mathcal{C}$, since a community clustering is determined up to communities permutation. For large numbers of communities C, since $|\mathcal{C}|=C!$, the above infimum becomes unpractical. Some solutions to this problem were proposed in [3].

Graph matching For this problem we assume that the available dataset consists of a sequence $\{((G_{i,1},G_{i,2},W_i),\Sigma_i)\}_{i=1}^m$ of graphs $G_{i,j}=(V_{i,j},E_{i,j})$, $j=1,2$, and permutation matrices Σ_i. Given two graphs G_1,G_2, the network output $\mathbf{f}(G_1,G_2;\Theta)\in\mathbb{R}^{N\times N}$ models the probabilities

$$f(G_1,G_2;\Theta)_{i,j} = P\{j=\sigma(i)\}, \quad \text{for } i,j\in V,$$

where σ is the permutation matching G_1 and G_2. The network is specified as a Siamese GCNN

$$\mathbf{z}_1 = \mathbf{x}^{(K)}(G_1;\Theta),\ \mathbf{z}_2 = \mathbf{x}^{(K)}(G_2;\Theta)\in\mathbb{R}^{N\times d},$$

where $\mathbf{x}^{(K)}(\cdot)$ is the K-th layer defined in Section 3.2.1. The output layer is then given by

$$\mathbf{y} = \mathbf{o}(\mathbf{z}_1, \mathbf{z}_2) = \mathrm{softmax}\big(\mathbf{z}_1 \mathbf{z}_2^T\big) \ ,$$

where the softmax function is applied row-wise. The input signal for the network $\mathbf{z}_i$ is taken to be the degree vector $\mathbf{x}_i^{(0)} = \mathrm{diag}(A_i \mathbf{1})$, where A_i is the adjacency matrix of the graph G_i, $i = 1, 2$. Such model is then evaluated with the following cross-entropy loss:

$$\ell(\mathbf{f}(G_1, G_2; \Theta); \Sigma) = -\sum_{i,j=1}^{N} \Sigma_{i,j} \log f(G_1, G_2; \Theta)_{i,j} \ .$$

Traveling salesman problem For this problem we assume that the available dataset consists of a sequence $\{((G_i, W_i), C_i)\}_{i=1}^m$ of graphs $G_i = (V_i, E_i)$ with weight matrices W_i and least cost cycles C_i (represented either as subsets of E_i or as permutations $\sigma_i \in S_N$). Given a graph G and a weight matrix W, the network output $\mathbf{f}(G, W; \Theta) \in \mathbb{R}^{N \times N}$ models the probabilities

$$f(G, W; \Theta)_{i,j} = P\{(i, j) \in C\}, \quad \text{for } i, j \in V,$$

where C denotes the least cost cycle. Here we assume the graph to be undirected and thus the adjacency matrix to be symmetric. The network $\mathbf{f}$ is specified by the architecture defined in Section 3.2.1, with output layer given by

$$\mathbf{y} = \mathbf{o}\big(\mathbf{x}^{(K)}; \eta\big) = \mathrm{softmax}\left(\frac{\mathbf{x}^{(K)} \big(\mathbf{x}^{(K)}\big)^T}{\big\|\mathbf{x}^{(K)}\big\|^2} - \eta I \right) \in \mathbb{R}^{N \times N} \ ,$$

the softmax function being applied row-wise. The parameter $\Theta^{(K+1)} = \eta$ avoids large diagonal influence. The input signal is taken to be the degree vector $\mathbf{x}^{(0)} = A\mathbf{1}$; in the inner layers, the weight matrix W is substituted to the adjacency matrix A. Such model is then evaluated with the following cross-entropy loss:

$$\begin{aligned} \ell(\mathbf{f}(G, W; \Theta); C) &= -\sum_{i=1}^{N} \log f(G, W; \Theta)_{i, \sigma(\sigma^{-1}(i)+1)} - \sum_{i=1}^{N} \log f(G, W; \Theta)_{i, \sigma(\sigma^{-1}(i)-1)} \\ &= 2 D_{KL}\left(\frac{1}{2} A_C \,\middle\|\, \mathbf{f}(G, W; \Theta) \right) + 2 \log 2 \ , \end{aligned}$$

where σ is the permutation associated with the cycle C and A_C is the (symmetric) matrix associated with C.

3.3 Numerics

The authors of [3, 39] assess the performances of the proposed methods with several numerical tests.

Community detection In [3], the authors tested the GNN architecture described above with $K = 30$ layers, $d = 10$ feature maps and $J = 3$, for graphs with $N = 1000$ vertices. AdaMax [29] with learning rate $\eta = 0.001$ was used to train. Different training set (depending on the path probabilities) were tested. For $k \leq 4$, the proposed GNN architecture was shown to be able to reach the information theoretic (IT) threshold and to outperform methods based on spectral clustering. The model was tested also for $k > 4$ (in particular for $k = 5$), where it is conjectured that a computational-to-statistical gap exists [8]. The model was shown to match the performances of loopy belief propagation algorithm. Also real datasets were tested, reporting SOA performances. For more details and other experiments, we refer to [3].

Graph matching In [39], the authors tested the GNN architecture described above with $K = 20$ layers, $d = 20$ feature maps, for graphs with $N = 50$ vertices. The GNN was tested to match perturbations of Erdos-Renyi and random regular graphs, according to the noise model considered in [14]. The network were trained using Adamax with learning rate $\eta = 0.001$ and batches of size 32. The architecture was shown to outperform both the SDP and the LowRankAlign methods. For more details on the experiments, we refer to [39].

Travelling salesman problem In [39], the authors tested the GNN architecture described above with $K = 40$ layers, $d = 80$ feature maps, for graphs with $N = 20$ vertices. Random graphs with weight based on Euclidean distance were generated; the ground truth cycles were generated using the Lin-Kernighan TSP Heuristic. The network were trained using Adamax with learning rate $\eta = 0.001$ and batches of size 32. The architecture was shown to obtain results comparable to the auto-regressive data-driven model [50] and Christofides [6]. For more details on the experiments, we refer to [39].

4 Conclusion

In this note, we reviewed the most important ideas to build neural network architectures on graphs. Certainly, there is much to be explored with GNN-architectures. For example, to our knowledge, there is no comprehensive comparison of all the models described in this note on a common benchmark. Also, there are several other NP hard inverse problems on graphs, where a data-driven approach might be fruitful, including finding Hamiltonian circuits, graph coloring, the vertex cover problem, or the K-center problem. We plan to revisit and conduct experiments to find GNN architectures suitable for these problems.

References

[1] Afonso S Bandeira. Ten lectures and forty-two open problems in the mathematics of data science, 2015.

[2] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in neural information processing systems*, pages 585–591, 2002.

[3] Joan Bruna and Xiang Li. Community detection with graph neural networks. *arXiv preprint arXiv:1705.08415*, 2017.

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

[5] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153–159, 1992.

[6] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[7] Adam Coates and Andrew Y. Ng. Selecting receptive fields in deep networks. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 2528–2536. Curran Associates, Inc., 2011.

[8] Aurelien Decelle, Florent Krzakala, Cristopher Moore, and Lenka Zdeborová. Asymptotic analysis of the stochastic block model for modular networks and its algorithmic applications. *Physical Review E*, 84(6):066106, 2011.

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

[10] Yash Deshpande and Andrea Montanari. Finding hidden cliques of size \sqrt {N/e} n/e in nearly linear time. *Foundations of Computational Mathematics*, 15(4):1069–1128, 2015.

[11] Inderjit S Dhillon, Yuqiang Guan, and Brian Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE transactions on pattern analysis and machine intelligence*, 29(11), 2007.

[12] Vincenzo Di Massa, Gabriele Monfardini, Lorenzo Sarti, Franco Scarselli, Marco Maggini, and Marco Gori. A comparison between recursive neural networks and graph neural networks. In *Neural Networks, 2006. IJCNN'06. International Joint Conference on*, pages 778–785. IEEE, 2006.

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

[14] Soheil Feizi, Gerald Quon, Muriel Medard, Manolis Kellis, and Ali Jadbabaie. Spectral alignment of networks. 2015.

[15] Paolo Frasconi, Marco Gori, and Alessandro Sperduti. A general framework for adaptive processing of data structures. *IEEE transactions on Neural Networks*, 9(5):768–786, 1998.

[16] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[18] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for learning in graph domains. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 729–734. IEEE, 2005.

[19] Karo Gregor and Yann LeCun. Emergence of complex-like cells in a temporal product network with local receptive fields. *arXiv preprint arXiv:1006.0448*, 2010.

[20] Markus Hagenbuchner, Alessandro Sperduti, and Ah Chung Tsoi. A self-organizing map for adaptive processing of structured data. *IEEE transactions on Neural Networks*, 14(3):491–505, 2003.

[21] David K Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011.

[22] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1994.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[24] Keld Helsgaun. *An effective implementation of K-opt moves for the Lin-Kernighan TSP heuristic*. PhD thesis, Roskilde University. Department of Computer Science, 2006.

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

[26] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[27] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[28] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the 20th international conference on machine learning (ICML-03)*, pages 321–328, 2003.

[29] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[30] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[31] Dan Kushnir, Meirav Galun, and Achi Brandt. Fast multiscale clustering and manifold identification. *Pattern Recognition*, 39(10):1876–1891, 2006.

[32] Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. 521:436–44, 05 2015.

[33] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[34] Jonathan Masci, Davide Boscaini, Michael Bronstein, and Pierre Vandergheynst. Geodesic convolutional neural networks on riemannian manifolds. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 37–45, 2015.

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

[36] Federico Monti, Michael Bronstein, and Xavier Bresson. Geometric matrix completion with recurrent multi-graph neural networks. In *Advances in Neural Information Processing Systems*, pages 3700–3710, 2017.

[37] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

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

[39] Alex Nowak, Soledad Villar, Afonso S Bandeira, and Joan Bruna. A note on learning algorithms for quadratic assignment with graph neural networks. *arXiv preprint arXiv:1706.07450*, 2017.

[40] A. Ortega, P. Frossard, J. Kovačević, J. M. F. Moura, and P. Vandergheynst. Graph signal processing: Overview, challenges, and applications. *Proceedings of the IEEE*, 106(5):808–828, May 2018.

[41] Panos M Pardalos, Henry Wolkowicz, et al. *Quadratic Assignment and Related Problems: DIMACS Workshop, May 20-21, 1993*, volume 16. American Mathematical Soc., 1994.

[42] Jiming Peng, Hans Mittelmann, and Xiaoxue Li. A new relaxation framework for quadratic assignment problems based on matrix splitting. *Mathematical Programming Computation*, 2(1):59–77, 2010.

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

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

[45] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. *arXiv preprint arXiv:1612.07659*, 2016.

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

[47] David I Shuman, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst. The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains. *IEEE Signal Processing Magazine*, 30(3):83–98, 2013.

[48] Alessandro Sperduti and Antonina Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

[49] Werner Uwents, Gabriele Monfardini, Hendrik Blockeel, Marco Gori, and Franco Scarselli. Neural networks for relational learning: an experimental comparison. *Machine Learning*, 82(3):315–349, 2011.

[50] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

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

[52] SHI Xingjian, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810, 2015.

[53] Lihi Zelnik-Manor and Pietro Perona. Self-tuning spectral clustering. In *Advances in neural information processing systems*, pages 1601–1608, 2005.