Graph Filters for Processing and Learning from Network Data

Elvin Isufi*, Bishwadeep Das, Alberto Natali, Mohammad Sabbaqi, Maosheng Yang

Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology
Delft, The Netherlands

This technical report contains the material for the graduate course given at Aalborg University on May 30 - June 1, June 2021. Last update: September 22, 2021. *Corresponding author: e.isufi-1@tudelft.nl.

Contents

1

Graph Representations

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is defined as a finite set of nodes $\mathcal{V} = \{1, 2, \ldots, N\}$, and a set of M edges $\mathcal{E}$ between these nodes. We denote an edge connecting nodes u and v by $(u, v) \in \mathcal{E}$. Figure 1.1 shows a graph with node set $\mathcal{V} = \{1, 2, 3, 4, 5\}$, and edge set $\mathcal{E} = \{(1,2), (1,5), (2,3), (2,5), (3,4), (4,5)\}$.

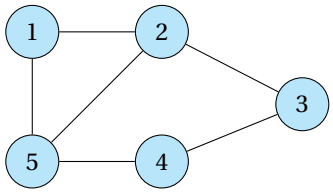


Figure 1.1: A graph with five nodes.

Undirected graph. A graph is undirected if the edge connecting node u to a node v also connects the node v to the node u, for all u and v. In other words, for an undirected graph, if $(u, v) \in \mathcal{E}$, then also $(v, u) \in \mathcal{E}$, as the example in Figure 1.1. The neighbourhood set of node u is defined as $\mathcal{N}_u := \{v | (u, v) \in \mathcal{E}\}$.

Directed graph. For directed graphs, edges have a direction. If edge $(u, v) \in \mathcal{E}$, then there is a directed edge leaving node u and incident into node v. In this case, node v is called an outneighbor of u, and u is called an inneighbor of v. For example, in Figure 1.2, there is a directed edge from node 2 to node 5, i.e., $(2,5) \in \mathcal{E}$, but $(5,2) \notin \mathcal{E}$. The local connectivity of a directed graph is captured by two neighboring sets, the set $\mathcal{N}_{u^{\text{in}}} := \{v | (v, u) \in \mathcal{E}\}$ which stands for the incoming neighborhood of node u and the outgoing neighborhood of node u, $\mathcal{N}_{u^{\text{out}}} = \{v | (u, v) \in \mathcal{E}\}$.

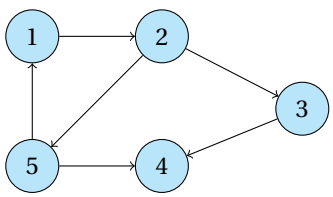


Figure 1.2: A directed graph with five nodes.

Weighted graphs. In general, the edges can also convey information about their strength, through a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ with a weight matrix $\mathbf{W} \in \mathbb{R}^{N \times N}$. For a weighted graph, if edge $(u, v) \in \mathcal{E}$ exists, then the entry (u, v) of $\mathbf{W}$ follows $W_{uv} > 0$; if there is no link between nodes u and v, then $W_{uv} = 0$. The weight matrix for an unweighted graph has entries $W_{uv} = 1$ if $(u, v) \in \mathcal{E}$ and $W_{uv} = 0$, otherwise. For an undirected graph, the weight matrix is symmetric, i.e., $\mathbf{W} = \mathbf{W}^\top$, while this does not hold for directed graphs.

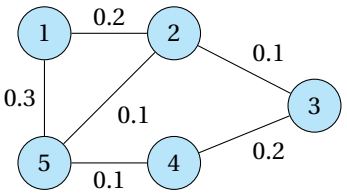


Figure 1.3: An undirected weighted graph with five nodes, where the values around the links are the edge weights.

The undirected weighted graph in Figure 1.3 can be represented by the weighted matrix

$$\mathbf{W} = \begin{bmatrix} 0 & 0.2 & 0 & 0 & 0.3 \\ 0.2 & 0 & 0.1 & 0 & 0.1 \\ 0 & 0.1 & 0 & 0.2 & 0 \\ 0 & 0 & 0.2 & 0 & 0.1 \\ 0.3 & 0.1 & 0 & 0.1 & 0 \end{bmatrix}.$$

1.1. Graph Matrices

Graphs can also be represented by algebraic matrices, for example, the adjacency matrix, the Laplacian matrix and the incidence matrix. We define them for both undirected and directed graphs.

Adjacency matrix. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can be formally represented by it adjacency matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$, which describes the node connectivity. For an unweighted graph, the entries of $\mathbf{A}$ are binary, i.e.,

$$A_{uv} = \begin{cases} 1, & \text{if } (u,v) \in \mathcal{E} \\ 0, & \text{if } (u,v) \notin \mathcal{E}. \end{cases} \tag{1.1}$$

For a weighted graph with weight matrix $\mathbf{W}$, the weight matrix $\mathbf{W}$ can be directly considered as its adjacency matrix $\mathbf{A}$, i.e.,

$$A_{uv} = \begin{cases} W_{uv}, & \text{if } (u,v) \in \mathcal{E} \\ 0, & \text{if } (u,v) \notin \mathcal{E}. \end{cases} \tag{1.2}$$

The adjacency matrix of an undirected graph is symmetric, i.e., $\mathbf{A} = \mathbf{A}^\top$. For a directed graph, the adjacency matrix is not symmetric, because $(u,v) \in \mathcal{E}$ does not mean (v,u) also exists. For example, the adjacency matrix of the undirected graph $\mathbf{A}_{\text{un}}$ in Figure 1.1, that of the directed graph $\mathbf{A}_{\text{dir}}$ in Figure 1.2, and that of the weighted undirected graph $\mathbf{A}$ in Figure 1.3 are respectively given by

$$\mathbf{A}_{\text{un}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \quad \mathbf{A}_{\text{dir}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \quad \mathbf{A} = \begin{bmatrix} 0 & 0.2 & 0 & 0 & 0.3 \\ 0.2 & 0 & 0.1 & 0 & 0.1 \\ 0 & 0.1 & 0 & 0.2 & 0 \\ 0 & 0 & 0.2 & 0 & 0.1 \\ 0.3 & 0.1 & 0 & 0.1 & 0 \end{bmatrix}.$$

Degree matrix. A degree matrix $\mathbf{D} = \text{diag}(d_1, \ldots, d_N)$ for an undirected graph is a diagonal matrix with u-th diagonal element d_u being the degree of node u. This is equal to the sum of weights of all edges connected to the node u,

$$d_u := \sum_{v=1}^{N} A_{uv}. \tag{1.3}$$

For an unweighted undirected graph, the value of d_u equals the number of edges connected to node u. For a directed graph, the number of edges leaving node u is the outdegree of u, denoted by d_u^{out}, and the number of edges incident into node u is its indegree and it is denoted by d_u^{in}.

For example, the degree matrices of the graphs in Figures 1.1, 1.2 and 1.3, are the following, respectively

$$\mathbf{D}_{\text{un}} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 3 \end{bmatrix} \quad \mathbf{D}_{\text{dir}}^{\text{in}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{D} = \begin{bmatrix} 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0.4 & 0 & 0 & 0 \\ 0 & 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0 & 0.5 \end{bmatrix}.$$

Normalized adjacency matrices. Given the degree matrix, another form of the adjacency matrix of a graph is the random-walk adjacency matrix, defined as

$$\mathbf{A}_{\mathrm{RW}} = \mathbf{D}^{-1}\mathbf{A}, \tag{1.4}$$

which is not symmetric in general. For numerical reasons, it is often advantageous to use the normalized adjacency matrix, defined as

$$\mathbf{A}_{\mathrm{n}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}. \tag{1.5}$$

Incidence matrix. We can also represent an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$ by its incidence matrix $\mathbf{B} \in \mathbb{R}^{N\times M}$. It describes the connections between nodes and edges of the graph. More specifically, the (i, j) entry describes the connection between node i and edge $j \in \mathcal{E}$, which is defined as follows

$$B_{ij} = \begin{cases} -1, & \text{if edge } j \text{ leaves from node } i, \\ 1, & \text{if edge } j \text{ is incident into node } i, \\ 0, & \text{otherwise.} \end{cases} \tag{1.6}$$

For example, the incidence matrix of the undirected graph $\mathbf{B}_{\mathrm{un}}$ in Figure 1.1, that of the directed graph $\mathbf{B}_{\mathrm{dir}}$ in Figure 1.2 are respectively given by (note that we order the edges in lexicographic order in the column direction)

$$\mathbf{B}_{\mathrm{un}} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \quad \mathbf{B}_{\mathrm{dir}} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \end{bmatrix}$$

Laplacian matrix. For an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$, the Laplacian matrix is defined as

$$\mathbf{L} := \mathbf{D} - \mathbf{A} = \mathbf{B}\mathbf{B}^{\top}, \tag{1.7}$$

where $\mathbf{D}$ is the degree matrix and $\mathbf{A}$ is the adjacency matrix. The diagonal elements of Laplacian matrix are the degree of each node, and the off-diagonal elements are nonpositive real numbers. The Laplacian matrices for the graphs in Figures 1.1 and 1.2, $\mathbf{L}_{\mathrm{un}}$ and $\mathbf{L}_{\mathrm{dir}}$ are, respectively

$$\mathbf{L}_{\mathrm{un}} = \begin{bmatrix} 2 & -1 & 0 & 0 & -1 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ -1 & -1 & 0 & -1 & 3 \end{bmatrix} \quad \mathbf{L}_{\mathrm{dir}} = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ 0 & 3 & -1 & 0 & -1 \\ 0 & 0 & 2 & -1 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ -1 & 0 & 0 & -1 & 3 \end{bmatrix}$$

Another form of the graph Laplacian is the so-called random-walk Laplacian, defined as

$$\mathbf{L}_{\mathrm{RW}} := \mathbf{D}^{-1}\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1}\mathbf{A}. \tag{1.8}$$

The random-walk graph Laplacian is rarely used for undirected graphs, since it loses the symmetry property of the original Laplacian. Again. for numerical reasons, it is often advantageous to use the normalized Laplacian matrix, defined as

$$\mathbf{L}_n = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2} \tag{1.9}$$

which is symmetric, and has all diagonal values unitary.

Graph shift operator. The graph shift operator (GSO) of a graph is a general algebraic representation of the graph. The GSO $\mathbf{S} \in \mathbb{R}^{N\times N}$ is a matrix whose entry S_{uv} is nonzero if $u = v$ or if $(u, v) \in \mathcal{E}$, zero, otherwise. The sparsity pattern of the matrix $\mathbf{S}$ captures the local structure of the graph. The graph matrices we saw in this section are choices of GSO.

1.2. Graph Spectra

In this section, we study the eigenvalues and eigenvectors of matrices associated with the graph. Graph spectral theory has resulted central in a variety of machine learning algorithms and the most popular one is the spectral clustering [4].

Graph shift operator. For undirected graph, the graph shift operator is always symmetric. Thus, it always enjoys the eigen-decomposition

$$\mathbf{S} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1} \tag{1.10}$$

with eigenvector matrix $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N]$ and diagonal eigenvalues matrix $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_N)$. For directed graphs, it is not always possible to do the eigen-decomposition of the graph shift operator. In these instances, the Jordan decomposition can be used but this is out of our scope. We refer readers to [6, 7]. For theoretical purposes, we will consider graphs for which the shift operator is always eigen-decomposition.

A spectral analysis for graphs can always be performed based on the graph Laplacian $\mathbf{L}$, since it is a symmetric matrix. The Laplacian matrix can be written as

$$\mathbf{L} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\top} \text{ or } \mathbf{L}\mathbf{U} = \mathbf{U}\mathbf{\Lambda}, \tag{1.11}$$

where $\mathbf{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \ldots, \lambda_N)$ is a diagonal matrix with the eigenvalues in an increasing order, and $\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_N]$ the orthonormal matrix of eigenvectors, with $\mathbf{U}^{-1} = \mathbf{U}^{\top}$. Then, every eigenvector $\mathbf{u}_k$, of the graph Laplacian $\mathbf{L}$ satisfies

$$\mathbf{L}\mathbf{u}_k = \lambda_k \mathbf{u}_k. \tag{1.12}$$

Here, we remark the important properties of the Laplacian eigen-decomposition:

- We order the eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_N$ in an increasing order. It is necessary to notice that the smallest eigenvalue of the graph Laplacian is $\lambda_1 = 0$, and the corresponding constant unit energy eigenvector is given by $\mathbf{u}_1 = [1, 1, \ldots, 1]^{\top}/\sqrt{N} = \mathbf{1}/\sqrt{N}$.
- The multiplicity of the eigenvalue λ_1 of the graph Laplacian is equal to the number of connected components (connected subgraphs) in the graph. For example, the graph in Figure 1.4 has two connected components, which corresponds to two eigenvalues of value zero [6].

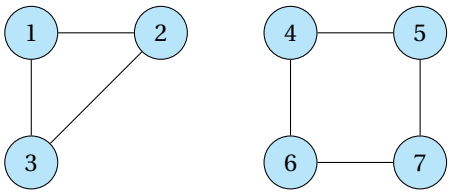


Figure 1.4: An example of a graph with two connected components..

Spectral clustering. Spectral clustering refers to a family of unsupervised learning algorithms, which is a fundamental tool in data mining. Given a set of N data points $\mathbf{x}_1, \ldots, \mathbf{x}_N$, the goal is to partition this set into k clusters. Spectral clustering is a graph-based algorithm for finding these k clusters in data. The technique involves representing the data in a low dimension given by a subset of the eigenvector of the Laplacian matrix.

We present here a popular spectral clustering method based on the normalized Laplacian [4]. Before using the spectral properties of Laplacian, we need to construct a graph from data points. First, compute weights $W_{ij} \geq 0$ that model the similarity between pairs of data points $(\mathbf{x}_i, \mathbf{x}_j)$. For instance, a common choice is the radial basis function kernel

$$W_{ij} = \exp(-\|\mathbf{x}_i - \mathbf{x}_j\|_2^2/\sigma^2), \tag{1.13}$$

where σ is an user-defined parameter. This gives rise to a graph $\mathcal{G}$ with N nodes and adjacency matrix $\mathbf{W} \in \mathbb{R}^{N\times N}$. Then, we can obtain the Laplacian matrix $\mathbf{L} = \mathbf{D} - \mathbf{W}$. Finally, we follow Algorithm 1 to find k clusters.

We applied Algorithm 1 on a stochastic block model graph. The graph has $N = 128$ nodes and $k = 5$ clusters with a 0.7 intra-cluster edge probability and 0.3 inter-cluster edge probability. We can see these results in Figures 1.5a and 1.5b, in which spectral clustering can identify the communities.

Algorithm 1 Spectral clustering [4]

The Laplacian matrix $\mathbf{L}$, the number of clusters k Compute the first k eigenvectors of $\mathbf{L}$: $\mathbf{U} = (\mathbf{u}_1, \mathbf{u}_1, \ldots, \mathbf{u}_k) \in \mathbb{R}^{N\times k}$.

Form the matrix $\mathbf{Y}_k \in \mathbb{R}^{N\times k}$ from $\mathbf{U}_k$ by normalizing each of $\mathbf{U}_k$'s rows to unit length: $[\mathbf{Y}_k]_{ij} = [\mathbf{U}_k]_{ij} / \sqrt{\sum_{j=1}^{k} [\mathbf{U}_k]_{ij}^2}$.

Treat each node i as a point in $\mathbb{R}^k$ by defining its feature vector $\mathbf{f}_i \in \mathbb{R}^k$ as the transposed i-th row of $\mathbf{Y}_k$: $\mathbf{f}_i := \mathbf{Y}_k^\top \boldsymbol{\delta}_i$, where $\delta_i(j) = 1$ if $j = i$ and 0 otherwise.

To obtain k clusters, run k-means with the Euclidean distance: $d_{ij} := \|\mathbf{f}_i - \mathbf{f}_j\|$.

(a) Data graph generated based on stochastic block model.

(b) Spectral clustering results.

Figure 1.5: Spectral clustering example.

1.3. Summary and Further Reading

In this chapter, we introduced the basics about graph theory, including the graph algebraic representations via adjacency matrices, incidence matrices and the graph Laplacians, which we call the graph shift operator. In addition, from the graph spectral domain, we can use the eigenvectors of the graph shift operator to perform data clustering, i.e., spectral clustering. For more details in graph theory and spectral clustering, we refer to the following literature [6, 1, 2, 3, 9, 5, 8, 10].

Bibliography

[1] Fan RK Chung and Fan Chung Graham. *Spectral graph theory*. 92. American Mathematical Soc., 1997.

[2] Stuart Lloyd. "Least squares quantization in PCM". In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[3] Antonio G Marques, Santiago Segarra, and Gonzalo Mateos. "Signal processing on directed graphs: The role of edge directionality when processing and learning from network data". In: *IEEE Signal Processing Magazine* 37.6 (2020), pp. 99–116.

[4] Andrew Ng, Michael Jordan, and Yair Weiss. "On spectral clustering: Analysis and an algorithm". In: *Advances in neural information processing systems* 14 (2001), pp. 849–856.

[5] Antonio Ortega et al. "Graph signal processing: Overview, challenges, and applications". In: *Proceedings of the IEEE* 106.5 (2018), pp. 808–828.

[6] Aliaksei Sandryhaila and José MF Moura. "Discrete signal processing on graphs". In: *IEEE transactions on signal processing* 61.7 (2013), pp. 1644–1656.

[7] Aliaksei Sandryhaila and José MF Moura. "Discrete signal processing on graphs: Frequency analysis". In: *IEEE Transactions on Signal Processing* 62.12 (2014), pp. 3042–3054.

[8] Jianbo Shi and Jitendra Malik. "Normalized cuts and image segmentation". In: *IEEE Transactions on pattern analysis and machine intelligence* 22.8 (2000), pp. 888–905.

[9] David I Shuman et al. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains". In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[10] Nicolas Tremblay and Andreas Loukas. "Approximating spectral clustering via sampling: a review". In: *Sampling Techniques for Supervised or Unsupervised Tasks* (2020), pp. 129–183.

2

Graph-Signal Coupling

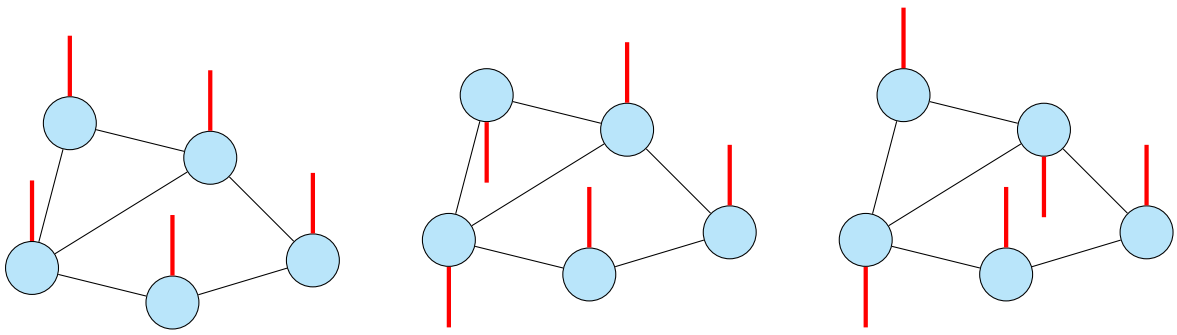
2.1. Signal-Topology Coupling

Graphs capture the relations between entities (nodes) and implicitly represent complex relations between the feature signals of these nodes. In this chapter, we investigate the coupling between such signal and the underlying topology.

2.1.1. Graph Signals

By assigning signal values to the nodes of a graph, we can define a signal on top of the graph, which we will also call a graph signal. Figure 2.1 illustrates three graph signals.

Definition 2.1.1 (Graph signal)**.** A graph signal is a mapping from the vertex set to the real set, i.e., $x_i : i \to \mathbb{R}$. For convenience, we collect all nodes' signals in the vector $\mathbf{x} \in \mathbb{R}^N$, where the ith component x_i of $\mathbf{x}$ is the signal value at node i.



(a) Constant signal. (b) Slow-varying signal. (c) High-varying signal.

Figure 2.1: Illustration of three different graph signals over the same undirected graph, where the red thick lines at the nodes indicate the signal values, up directions corresponding to positive values and down directions corresponding to negative values. From left to right, the signal variation over the graph increases.

Definition 2.1.2 (Multivariate graph signal)**.** If we have F graph signals defined on the graph, we can stack them column by columnwise into an $N \times F$ matrix, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_F]$, where the f-th column of $\mathbf{X}$ is the f-th graph signal. We will refer to these values as a multivariate signal.

The multivariate graph signal can model a number F of N-dimensional graph signals. For example, in a weather sensor network, we collect multiple sensor measurements including temperature, pressure, humidity. In this chapter, we will focus on the univariate graph signal.

2.1.2. Signal Diffusion Over Graphs

The edges of a graph play a role on how signal values diffuse to neighboring nodes. Such operation is called graph signal shifting and it is defined as

$$\mathbf{x}^{(1)} = \mathbf{S}\mathbf{x} \tag{2.1}$$

where $\mathbf{x}^{(1)}$ stands for the one-shift of a graph signal $\mathbf{x}$ by a graph shift operator $\mathbf{S}$. Similarly, $\mathbf{x}^{(0)} = \mathbf{I}_N\mathbf{x}$ can be considered as the zero-shifting of $\mathbf{x}$ over the graph, i.e., the graph signal itself. Graph signal shifting (2.1) is a local operation. In fact, at certain node i, the signal shifting reduces to

$$x_i^{(1)} = \sum_{j\in\mathcal{N}_i\cup\{i\}} S_{ij}x_j \tag{2.2}$$

where $\mathcal{N}_i$ is the neighborhood of node i. In (2.2), node i collects signal information only from its direct neighbors $\mathcal{N}_i$ and itself, without requiring the global information about the signal $\mathbf{x}$. In the following, we detail the shifting operation when graph shift operator $\mathbf{S}$ is graph Laplacian and adjacency matrix, respectively.

Adjacency matrix. For $\mathbf{S} = \mathbf{A}$, the one-shift operation (2.1) becomes $\mathbf{x}^{(1)} = \mathbf{A}\mathbf{x}$, while the operation on node i is

$$x_i^{(1)} = \sum_{j\in\mathcal{N}_i} A_{ij}x_j, \tag{2.3}$$

where A_{ij} is the entry (i,j) of adjacency matrix $\mathbf{A}$. This operation takes the signals from the neighbors and linearly combines them by the edge weights. In Figure 2.2, we illustrate the one-step shifting operation of an all-ones graph signal on node 2.

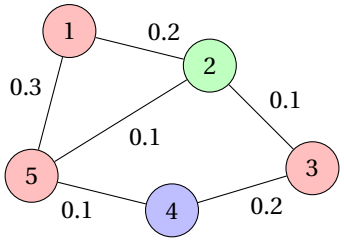


Figure 2.2: The shifting operation illustration. The green node is the target node, and the red ones are its neighbours, which exchange their signal information to node 2. If an all-ones signal is on the graph, then the signal value at node 2 after one-step shifting by the adjacency matrix is $0.2\cdot 1 + 0.1\cdot 1 + 0.1\cdot 1 = 0.4$.

Laplacian matrix. For $\mathbf{S} = \mathbf{L}$, the one-step shifting operation (2.1) becomes $\mathbf{x}^{(1)} = \mathbf{L}\mathbf{x}$ and the shifting on node i works as follows

$$x_i^{(1)} = \sum_{j\in\mathcal{N}_i} A_{ij}(x_i - x_j). \tag{2.4}$$

From a node perspective, this operation also takes the information from the node neighbors and linearly combines them with the node's own state. In Figure 2.3, we illustrate the operation (2.4) of an all-ones graph signal on node 2.

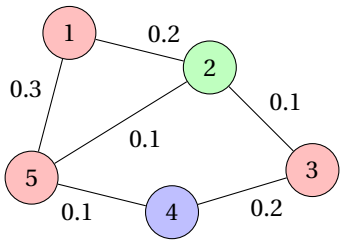


Figure 2.3: The shifting operation illustration. The green node is the target node, and the red ones are its neighbours, which exchange their signal information to node 2. If an all-ones signal is on the graph, then the signal value at node 2 after one-step shifting by the graph Laplacian matrix is $(0.2+0.1+0.1)\cdot 1 - (0.2\cdot 1 + 0.1\cdot 1 + 0.1\cdot 1) = 0$. This is expected as a constant vector is an eigenvector of $\mathbf{L}$ corresponding to zero eigenvalues.

Note that if the graph topology changes, the signal diffusion will change too. This is because the neighboring set $\mathcal{N}_i$ and the entries S_{ij} will be different. For example, suppose in Figure 2.3, edge $(2,5)$ is removed. Then,

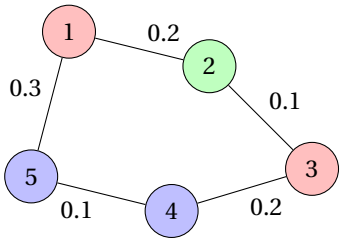


Figure 2.4: The shifting operation illustration. The green node is the target node, and the red ones are its neighbours (node 5 is no longer a neighbor), which exchange their signal information to node 2. If an all-ones signal is on the graph, then the signal value at node 2 after one-step shifting by the graph Laplacian matrix is $(0.2+0.1)\cdot 1-(0.2\cdot 1+0.1\cdot 1)=0$. This is expected as a constant vector is an eigenvector of $\mathbf{L}$ corresponding to zero eigenvalues.

the shifting operation (2.3) on node 2 will not anymore account for the signal value on node 5 but only on nodes 1 and 3.

The diffusion over graphs of the signal $\mathbf{x}$ can be achieved by the matrix-vector multiplication $\mathbf{Sx}$, with the graph shift operator $\mathbf{S}$ as in (2.1). This operation is a distributed operation as what we indicated in (2.2) from a node perspective. For instance, in a sensor network, this can be done by local information communication between sensors. It has a complexity of $\mathcal{O}(M)$, since the number of nonzeros in the graph shift operator $\mathbf{S}$ is M, i.e., the number of edges.

Remark. [Shifting multivariate graph signals.] The graph shifting operation can be extended to the multivariate case. Consider a multivariate graph signal $\mathbf{X}=[\mathbf{x}_1,\mathbf{x}_2,\ldots,\mathbf{x}_F]\in\mathbb{R}^{N\times F}$ defined on a graph with graph shift operator $\mathbf{S}$. The one-step shifting can be done as

$$\mathbf{X}^{(1)}=\mathbf{SX}=[\mathbf{Sx}_1,\mathbf{Sx}_2,\ldots,\mathbf{Sx}_F]=[\mathbf{x}_1^{(1)},\mathbf{x}_2^{(1)},\ldots,\mathbf{x}_F^{(1)}], \tag{2.5}$$

which is a concatenation of the univariate graph signal shifting case.

2.2. Signal Variation Over the Graph

To develop a spectral representation for graph signals, we need first to characterize their variation over the graph. In this section, we analyze how signal changes over the graph, which is important to the generalization of signal spectral analysis from time domain to graph domain. In turn, to characterize the signal variation we need to use a specific metric. The literature distinguished different metrics to characterize the signal variation over the graph depending if the graph is undirected [10] or directed [8].

2.2.1. Undirected Graph

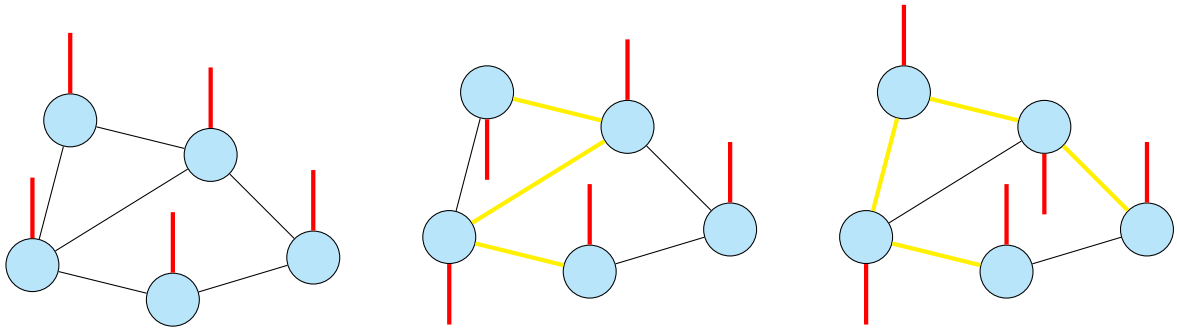
Consider the scenario in Figure 2.5, depicting three different graph signals residing on the same topology. The vertical bars indicate the signal value where positive values are illustrated with a bar oriented upwards and a negative value with a bar oriented downwards. We are interested in finding which of the three graph signals varies the most and which the least. Visually, we can see that the signals in (b) and (c) vary more than the constant signal in (a).

One way to quantify the signal variation over the graph is to count the number of times that the graph signal changes sign between nodes that share an edge. For the signal in (a), we have no sign change, for the signal in (b), we have three sign changes, and for the signal in (c), we have four sign changes. Thus, we may "more formally" conclude that the signal in (a) is the least varying over the graph and signal in (c) is the most varying over the graph.

Graph signal gradient. Formally, for a signal $\mathbf{x}$ over a graph $\mathcal{G}=(\mathcal{V},\mathcal{E})$, the variation of $\mathbf{x}$ with respect to the edge $e_{i,j}=(v_i,v_j)$ valued at vertex v_i is given by the edge derivative

$$\frac{\partial \mathbf{x}}{\partial e_{i,j}}\Big|_{v_i}=\sqrt{A_{i,j}}(x_i-x_j), \tag{2.6}$$

that is, the difference of the signal at the end nodes of edge $e_{i,j}$ weighted by the square root of the edge weight. The edge derivative quantifies how much signal values change in two connecting nodes. The smaller the derivative, the more similar the signal values x_i and x_j. Subsequently, the graph gradient of signal $\mathbf{x}$ evaluated



(a) Constant signal. (b) Slow-varying signal. (c) High-varying signal.

Figure 2.5: Illustration of three different graph signals over the same undirected graph, where the red thick lines at the nodes indicate the signal values, up directions corresponding to positive values and down directions corresponding to negative values. The yellow edge between nodes indicates there is a sign transition between two adjacent nodes. From left to right, we see the signal variation over the graph increases, as in (a) there is no sign transition, in (b), three sign transitions and four in (c).

at vertex v_i is the vector

$$\nabla_{v_i}\mathbf{x} = \left\{ \frac{\partial \mathbf{x}}{\partial e_{i,j}}\bigg|_{v_i} \right\}_{e_{i,j}\in\mathcal{E}\ s.t.\ e_{i,j}=(v_i,v_j)\ \text{for some}\ v_j\in\mathcal{V}} \tag{2.7}$$

containing all partial derivatives of $\mathbf{x}$ at node v_i. The graph gradient in (2.7) is a local measure and indicates not the directions on which the signal changes the least or the most a node v_i. Then, the ℓ_2-norm of (2.7)

$$\|\nabla_{v_i}\mathbf{x}\|_2 = \Big(\sum_{v_j\in\mathcal{N}_i} A_{i,j}(x_i - x_j)^2 \Big)^{\frac{1}{2}} \tag{2.8}$$

provides a measure of signal variability of $\mathbf{x}$ around vertex v_i with respect to its neighbors $\mathcal{N}_i$. Precisely, the graph signal is said to be smooth (i.e., vary little) in the v_i's neighborhood $\mathcal{N}_i$ if $\|\nabla_{v_i}\mathbf{x}\|_2$ is small, or equivalently if x_i is similar to x_j for $v_j \in \mathcal{N}_i$. Inequality $\|\nabla_{v_i}\mathbf{x}\|_2 < \|\nabla_{v_j}\mathbf{x}\|_2$ indicates that the signal variation at node v_j is higher than the signal variation at node v_i.

Dirichlet form. The notion of signal variation can be extrapolated from a particular node to the whole graph by means of the p-Dirichlet form of $\mathbf{x}$

$$S_p(\mathbf{x}) = \frac{1}{p}\sum_{v_i\in\mathcal{V}} \|\nabla_{v_i}\mathbf{x}\|_2^p = \frac{1}{p}\sum_{v_i\in\mathcal{V}} \Big(\sum_{v_j\in\mathcal{N}_i} A_{i,j}(x_i - x_j)^2 \Big)^{\frac{p}{2}} \tag{2.9}$$

which consists of a weighted sum of the signal variations in all nodes. In other words, it is the sum of the gradient at all nodes. Particular forms of $S_p(\mathbf{x})$ are:

- $S_1(\mathbf{x})$ referred to as the signal total variation [10, 5, 8]

$$S_1(\mathbf{x}) = \sum_{v_i\in\mathcal{V}} \|\nabla_{v_i}\mathbf{x}\|_2 = \sum_{v_i\in\mathcal{V}} \Big(\sum_{v_j\in\mathcal{N}_i} A_{i,j}(x_i - x_j)^2 \Big)^{\frac{1}{2}}. \tag{2.10}$$

- $S_2(\mathbf{x})$ referred to as the graph Laplacian quadratic form [10, 5, 8]

$$S_2(\mathbf{x}) = \frac{1}{2}\sum_{v_i\in\mathcal{V}} \|\nabla_{v_i}\mathbf{x}\|_2^2 = \frac{1}{2}\sum_{v_i\in\mathcal{V}} \Big(\sum_{v_j\in\mathcal{N}_i} A_{i,j}(x_i - x_j)^2 \Big) = \mathbf{x}^\top \mathbf{L}\mathbf{x}. \tag{2.11}$$

  Typically, the 2-Dirichlet form $S_2(\mathbf{x})$ is used to measure the signal variation over undirected graphs, also known as signal smoothness measure. If it is small, the signal changes slowly over the graph. Contrarily, it is large, if the signal changes rapidly over the graph. For example in Figure 2.5, $S_2(\mathbf{x}) = 0$ for the signal in (a) (the signal has no variation) and $S_2(\mathbf{x})$ of the signal in (b) is smaller than $S_2(\mathbf{x})$ in (c).

2.2.2. Directed Graph

For signals on a general directed graph, we can generalize the total variation (TV) of a discrete signal in classical DSP to graphs [8, 9]. The TV of a discrete signal is defined as the sum of magnitudes of differences

$$\mathrm{TV}(\mathbf{x}) = \sum_n |x_n - x_{n-1}|. \tag{2.12}$$

The total variation above for time series or space signals, such as images, compares how the signal varies with time or space. More specifically, it compares a signal $\mathbf{x}$ to its shifted version: the smaller the difference between the original signal and the shifted one, the lower the signal's variation. Thus, the signal total variation on graphs ($\mathrm{TV}_{\mathscr{G}}$) of a graph signal $\mathbf{x}$ is defined as

$$\mathrm{TV}_{\mathscr{G}}(\mathbf{x}) = \|\mathbf{x} - \mathbf{A}_n\mathbf{x}\|_1, \tag{2.13}$$

where the term $\mathbf{A}_n\mathbf{x}$ is the one-shift signal of $\mathbf{x}$. The definition uses the normalized adjacency matrix $\mathbf{A}_n$ to guarantee that the shifted signal is properly scaled for comparison with the original signal. If the total variation of a graph signal is large, then the signal changes rapidly over the graph; otherwise, it changes slowly.

Remark. Other forms of signal variation measures are possible. We refer to [9] for a detailed discussion. Note that the definition of signal variation is mainly used to generalize the definition of frequency from classical DSP to graph signal processing. But different definitions all lead to the same frequency ordering in graph domain [9].

2.3. Regularization on Graphs

The variations measures in (undirected) and (directed) show explicitly how the graph signal is tied with the underlying topology. We can, therefore, use such variation as a prior information to regularize the signal. A typical application is to retrieve the original graph signal from noisy measurements. Let $f(\mathbf{x};\mathbf{S})$ be a measure that captures the coupling between the signal $\mathbf{x}$ and the graph $\mathscr{G}$ with shift operator $\mathbf{S}$ (e.g., the signal variation discussed in Section 2.2) and let $\mathbf{y}$ be the measurements for signal $\mathbf{x}$. We can then estimate the signal by solving the optimization problem

$$\hat{\mathbf{x}} = \underset{\mathbf{x}\in\mathbb{R}^N}{\operatorname{argmin}} \|\mathbf{x} - \mathbf{y}\|_2^2 + \omega_0 f(\mathbf{x};\mathbf{S}). \tag{2.14}$$

The first term in (2.14) $\|\mathbf{x}-\mathbf{y}\|_2^2$ is the error fitting term, which looks for a solution to minimize the difference between the signal $\mathbf{x}$ and the observation $\mathbf{y}$. The second term, $f(\mathbf{x};\mathbf{S})$, is called regularization term, which serves as a penalty function to induce some prior about the graph signal $\mathbf{x}$ in the final estimate $\hat{\mathbf{x}}$. The regularization weight $\omega_0 > 0$ controls the trade-off between the error fitting term and regularization term. For $\omega \to 0$, we have solution (2.14) uses little the prior information $f(\mathbf{x};\mathbf{S})$ and aims to fit the signal to the observation $\mathbf{y}$. This is typically the case when there is little noise in the signal. Contrarily, if $\omega \to \infty$, we give in (2.14) more weight to the prior information and less to the observation. This may be the cases when the measurement $\mathbf{y}$ is heavily corrupted by noise.

In practice, different regularizers can be chosen, such as the variants of the graph Laplacian quadratic form, total variation of signals over graphs, or their variants. The most common one is the signal smoothness measure $F(\mathbf{x};\mathbf{L}) = \mathbf{x}^\top\mathbf{L}\mathbf{x}$, which assumes that the graph signal is smooth over the graph. We detail the latter next. Readers more interested in the subject may refer, e.g., [11, 7], for other regularization terms.

2.3.1. Global Regularization

Suppose we have a graph signal $\mathbf{x} \in \mathbb{R}^N$ defined over an undirected graph with the Laplacian $\mathbf{L}$, and we have collected the observations $\mathbf{y} \in \mathbb{R}^P$ as

$$\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{n}, \tag{2.15}$$

where matrix $\mathbf{C} \in \{0,1\}^{P\times N}$ is a binary matrix and $\mathbf{n} \in \mathbb{R}^P$ is the additive Gaussian noise with zero mean and a covariance matrix $\mathbf{\Sigma}$. Our task now is to estimate the values of $\mathbf{x}$ on all nodes from the partial observation $\mathbf{y}$. We can use expression (2.14) and recover the signal $\mathbf{x}$ by solving

$$\hat{\mathbf{x}} = \underset{\mathbf{x}\in\mathbb{R}^N}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{C}\mathbf{x}\|_2^2 + \omega_0 f(\mathbf{x};\mathbf{L}). \tag{2.16}$$

A typical prior about natural graph signal is that their quadratic form $S_2(\mathbf{x}) = \mathbf{x}^\top\mathbf{L}\mathbf{x}$ is small. This is the case for signal values that are similar in adjacent nodes. We can therefore use such prior to estimate $\mathbf{x}$ and rephrase

(2.16) as

$$\hat{\mathbf{x}} = \underset{\mathbf{x}\in\mathbb{R}^N}{\operatorname{argmin}} \|\mathbf{y}-\mathbf{C}\mathbf{x}\|_2^2 + \omega_0 \mathbf{x}^\top \mathbf{L}\mathbf{x} \tag{2.17}$$

which is also known as Tikhonov regularization and graph Laplacian regularization [10]. In problem (2.17), we will typically give a large ω_0 when $S_2(\mathbf{x})$ is small and vice versa. This is because, the larger ω_0, the more importance we give to the regularizer than to the error fitting term. Note that problem (2.17) is convex and has the closed form solution

$$\hat{\mathbf{x}} = (\mathbf{C}^\top\mathbf{C} + \omega_0\mathbf{L})^{-1}\mathbf{C}^\top\mathbf{y} \tag{2.18}$$

where matrix $\mathbf{C}^\top\mathbf{C} + \omega_0\mathbf{L}$ is positive definite and invertible [2]. This is known as the graph signal interpolation problem.

When the sampling matrix $\mathbf{C} = \mathbf{I}$ and the observation $\mathbf{y} \in \mathbb{R}^N$ is obtained from all the nodes, problem (2.17) boils down to a simple denoising problem and the estimate signal has the form

$$\hat{\mathbf{x}} = (\mathbf{I} + \omega_0\mathbf{L})^{-1}\mathbf{y}. \tag{2.19}$$

The graph Laplacian regularizer $\mathbf{x}^\top\mathbf{L}\mathbf{x}$ guarantees that the solutions (2.18) and (2.19) will be smooth over the graph.

However, we should consider that the regularizer term adds a bias to the solution. The bias $\mathbf{b}(\omega_0)$ of the denoising estimate (2.19) is

$$\mathbf{b}(\omega_0) = \mathbb{E}[\hat{\mathbf{x}}] - \mathbf{x} = ((\mathbf{I} + \omega_0)^{-1} - \mathbf{I})\mathbf{x}. \tag{2.20}$$

On the other hand, the variance can be found as

$$\operatorname{var}(\omega_0) = \mathbb{E}[\|\hat{\mathbf{x}} - \mathbb{E}[\hat{\mathbf{x}}]\|^2] = \operatorname{tr}((\mathbf{I} + \omega_0\mathbf{L})^{-2}\Sigma). \tag{2.21}$$

where $\operatorname{tr}(\cdot)$ is the trace operator. By summing the squared bias and variance, we have the mean squared error (MSE) of the estimate $\hat{\mathbf{x}}$

$$\operatorname{mse}(\omega_0) = \mathbb{E}[\|\hat{\mathbf{x}} - \mathbf{x}\|^2] = \|\mathbf{b}(\omega_0)\|^2 + \operatorname{var}(\omega_0) = \operatorname{tr}(((\mathbf{I} + \omega_0)^{-1} - \mathbf{I})^2\mathbf{x}\mathbf{x}^\top) + \operatorname{tr}((\mathbf{I} + \omega_0\mathbf{L})^{-2}\Sigma), \tag{2.22}$$

which quantifies how well we estimated the signal. Notice that from (2.22) that ω_0 will now balance the bias term with the variance. For different values of ω_0, we may have a lower bias in the solution and a higher variance and vice verse. A more complete bias-variance trade-off study can be found in [3].

2.3.2. Local Regularization

The problem in (2.14) has a global regularizer, meaning that the signal fitting component on each node is penalized equally. For instance, problem (2.17) penalizes the global signal smoothness instead of the variability in the surrounding of a node. In turn, this strategy cannot distinguish the local signal-topology coupling. For instance, for signals that are piecewise-smooth or piecewise-constant [11], we want to have different penalty terms in different components of the graph. To account for this local variability, we discuss next a node-adaptive regularizer to perform graph signal reconstruction in which each nodal signal is penalized differently.

Consider the signal model in (2.15) with matrix $\mathbf{C} = \mathbf{I}$. The node-adaptive regularization is formulated as

$$\hat{\mathbf{x}}(\boldsymbol{\omega}) = \underset{\mathbf{x}\in\mathbb{R}^N}{\operatorname{argmin}} \|\mathbf{x}-\mathbf{y}\|_2^2 + (\operatorname{diag}(\boldsymbol{\omega})\mathbf{x})^\top\mathbf{L}(\operatorname{diag}(\boldsymbol{\omega})\mathbf{x}), \tag{2.23}$$

where the vector parameter $\boldsymbol{\omega} = [\omega_1, \omega_2, \ldots, \omega_N] \in \mathbb{R}^N$ imposes a different penalty on different nodes. Expanding the node-adaptive regularizer, we have the quadratic form

$$(\operatorname{diag}(\boldsymbol{\omega})\mathbf{x})^\top\mathbf{L}(\operatorname{diag}(\boldsymbol{\omega})\mathbf{x}) = \frac{1}{2}\sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{N}_i} A_{ij}(\omega_i x_i - \omega_j x_j)^2. \tag{2.24}$$

This form can be seen as follows: first, it re-weights each signal entry x_i with a parameter ω_i and then computes the regular Laplacian quadratic measure. For the nodes with signal values close to each other, we can assign them similar weights to measure the local signal variation, thus, to promote the local signal smoothness. The closed form solution of (2.23) is

$$\hat{\mathbf{x}}(\boldsymbol{\omega}) = (\mathbf{I} + \operatorname{diag}(\boldsymbol{\omega})\mathbf{L}\operatorname{diag}(\boldsymbol{\omega}))^{-1}\mathbf{y} := \mathbf{H}(\boldsymbol{\omega})\mathbf{y}, \tag{2.25}$$

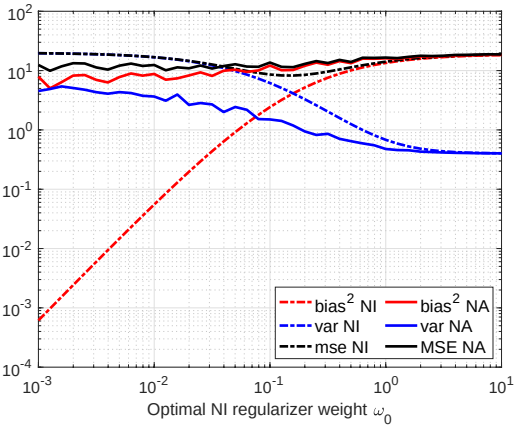


Figure 2.6: Bias-variance trade-off comparison between the estimate (2.19) and the node-adaptive estimate (2.25) [12], where NI stands for node-invariant, i.e., the estimate (2.19) by the graph Laplacian regularizer.

which can be implemented similarly as the solution (2.19) since the term $\text{diag}(\boldsymbol{\omega})\mathbf{L}\text{diag}(\boldsymbol{\omega})$ shares the sparsity with the graph Laplacian. In [12], the bias-variance tradeoff for the node-adaptive estimate (2.25) has been studied. The MSE of the estimate (2.25) is

$$\text{mse}(\boldsymbol{\omega}) = \|\mathbf{b}(\boldsymbol{\omega})\|_2^2 + \text{var}(\boldsymbol{\omega}) = \text{tr}((\mathbf{I}-\mathbf{H}(\boldsymbol{\omega}))^2\mathbf{x}\mathbf{x}^\top) + \text{tr}(\mathbf{H}^2(\boldsymbol{\omega})\boldsymbol{\Sigma}) \tag{2.26}$$

with bias

$$\mathbf{b}(\boldsymbol{\omega}) = ((\mathbf{I}+\text{diag}(\boldsymbol{\omega})\mathbf{L}\text{diag}(\boldsymbol{\omega}))^{-1} - \mathbf{I})\mathbf{x} \tag{2.27}$$

and variance

$$\text{var}(\boldsymbol{\omega}) = \text{tr}((\mathbf{I}+\text{diag}(\boldsymbol{\omega})\mathbf{L}\text{diag}(\boldsymbol{\omega}))^{-2}\boldsymbol{\Sigma}). \tag{2.28}$$

From above the bias-variance trade-off, we see that when entries of $\boldsymbol{\omega}$ are close to zero, the bias is low and the MSE is governed by a high variance. Instead, if the entries of $\boldsymbol{\omega}$ are large, the bias is large and dominates the MSE. Compared to the global regularizer, the node-adaptive estimate gives rise to an improved reconstruction performance.

Lemma 2.3.1 ([12])**.** Consider the graph Laplacian regularizer estimate (2.19) and the node-adaptive estimate (2.25). The variances for these two estimates would follow $\text{var}(\boldsymbol{\omega}) \leq \text{var}(\omega_0)$, if the node-adaptive weights $\boldsymbol{\omega} = [\omega_1, \omega_2, \ldots, \omega_N]^\top$ satisfy

$$\omega_0 \leq \omega_i^2, \text{ for } i = 1, 2, \ldots, N. \tag{2.29}$$

The authors in [12] has shown the bias-variance trade-off for the reconstruction estimates (2.19) and (2.25) by requiring the Lemma 2.3.1 as in Figure 2.6. We can see that the variance of the node-adaptive estimate is consistently smaller than the graph Laplacian estimate (2.19). More importantly, a smaller MSE of the node-adaptive estimate can be obtained in some cases. This is also provided in [12], but we will not detail here.

2.4. Applications

We now apply the above concepts to three real-world applications, namely, collaborative filtering in recommender systems [4], blog classification [9] and temperature data denoising and interpolation [12].

2.4.1. Collaborative Filtering for Rating Prediction

In recommender systems, we have a set of users $\mathcal{U} = \{u_1, \ldots, u_U\}$ (with cardinality $|\mathcal{U}| = U$) interacting with the set of items $\mathcal{I} = [i_1, \ldots, i_I]$ (with cardinality $|\mathcal{I}| = I$) for rating prediction by giving a rating score, e.g., a number from one to five. This setting is represented with the user-item matrix (UIM) $\mathbf{X} \in \mathbb{R}^{U \times I}$, where the rating user u gives to item i is the entry X_{ui}. See an example of rating matrix in Table 2.1. In practice, the rating matrix shows high sparsity. The goal in recommender systems is to use the known ratings to predict the missing values in the rating matrix.

Collaborative filtering is the most used algorithm to do rating prediction [4]. We can distinguish two types of collaborative filtering, user-based and item-based. In this section, we will discuss the user-based collaborative

	Item 1	Item 2	Item 3	Item 4	Item 5
User 1	3	4	-	5	1
User 2	3	-	2	4	-
User 3	5	2	2	-	5

Table 2.1: User-item rating matrix, where "-" indicates the user did not rate the item.

filtering. The item-based collaborative filtering follows a similar procedure. User-based collaborative filtering relies on the fact that similar users tend to give similar ratings to items. Thus, by evaluating the similarity between user-pairs, we can predict ratings based on one user's neighbors, i.e., its similar users. This is the core of user-based collaborative filtering. Similarly, item-based uses the similarity between item-pairs.

Given a symmetric user-user similarity matrix $\mathbf{B} \in \mathbb{R}^{U\times U}$ which contains the similarity scores between every user-user pair. Here, we consider the Pearson correlation as the similarity measure. Given two users u and v, let $\mathscr{I}_{uv}$ be the set of items rated by both u and v. Then, we compute the average rating that user u gives to items $i \in \mathscr{I}_{uv}$ as

$$\mu_u = \frac{1}{|\mathscr{I}_{uv}|} \sum_{i\in\mathscr{I}_{uv}} X_{ui}. \tag{2.30}$$

The Pearson correlation coefficient B_{uv} between users u and v is

$$B_{uv} = \frac{\sum_{i\in\mathscr{I}_{uv}}(X_{ui}-\mu_u)(X_{vi}-\mu_v)}{\sqrt{\sum_{i\in\mathscr{I}_{uv}}(X_{ui}-\mu_u)^2 \sum_{i\in\mathscr{I}_{uv}}(X_{vi}-\mu_v)^2}}. \tag{2.31}$$

The coefficient B_{uv} measures the similarity between users u and v based on the items they both interact. The denominator normalizes the correlation coefficients such that $B_{uv} \in [-1,1]$ for all user-user pairs. If users u and v have similar ratings for common items, the correlation will be close to one; if they have an opposite rating pattern, the correlation will be close to negative one.

Let $\mathscr{N}_{ui}$ be the set of k users set which are most similar to user u and rated item i, i.e., the k largest entries of the u-th row in matrix $\mathbf{B}$. We can predict the rating of user u to item i as

$$\hat{X}_{ui} = \frac{\sum_{v\in\mathscr{N}_{ui}} B_{uv} X_{vi}}{\sum_{v\in\mathscr{N}_{ui}} B_{uv}}. \tag{2.32}$$

In the vector form, expression (2.32) reads as

$$\hat{\mathbf{x}}_i = \mathbf{B}_i \mathbf{x}_i \tag{2.33}$$

where matrix set $\mathbf{B}_i \in \mathbb{R}^{U\times U}$ is the adjacency matrices of a user-user graph with respect to item i and has entries

$$[\mathbf{B}_i]_{uv} = \begin{cases} B_{uv} / \sum_{v'\in\mathscr{N}_{ui}} B_{uv'} & \text{if } v \in \mathscr{N}_{ui}, \\ 0 & \text{if } v \notin \mathscr{N}_{ui}. \end{cases} \tag{2.34}$$

In this way, matrix $\mathbf{B}_i$ describes the similarity graph between users for the i-th item, and the rating vector $\mathbf{x}_i$ is the respective as a graph signal. Then, we can see the collaborative filtering operation (2.33) as a graph signal shifting operation [cf. (2.1)]. This is the interpretation of collaborative filtering from a graph signal processing perspective [4].

Here we report a simulation based on graph shifting operation for rating prediction on the MovieLens 100k dataset [6], which contains ratings from 943 users on 1682 movies. There are 100,000 available ratings ranging from 1 to 5. To predict the unknown ratings, user-based collaborative filter (2.33) and its extended version with more than one-step shifting, i.e., based on graph filters, $\hat{\mathbf{x}}_i = \sum_{k=1}^{K} h_k \mathbf{B}_i^k \mathbf{x}_i$ are used [8, 4]. The similarity matrices representing the user-user similarity graphs, and filter coefficients are trained on the training set which contains 99,900 ratings. For performance evaluation, the root mean squared error (RMSE) is used on the testing data with 100 ratings. The results are reported in Table 2.2.

From the results, we observe that when the order of shifting is higher, the RMSE is getting smaller and the prediction is more accurate. This is because with more than one-step shifting, the one-hop further users with similar taste also contribute to the rating prediction. However, the performance will not be improved further upto certain number of shifting because the neighbors far away with similar taste probably will not provide new information. For a more detailed discussion regarding the rating prediction based on graph filters, we refer to [4].

Number of shifting	1	2	3	4	5
RMSE	0.9036	0.8921	0.8226	0.8218	0.8128

Table 2.2: Graph signal processing for rating prediction.

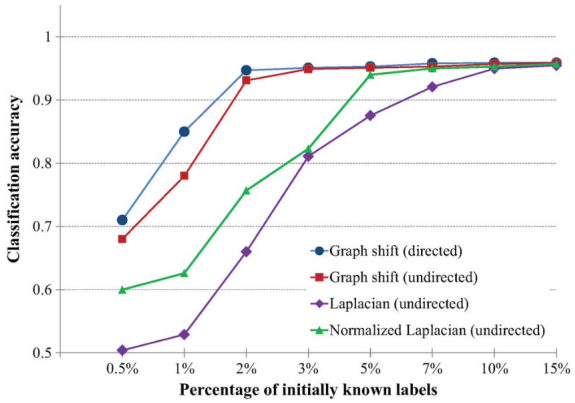


Figure 2.7: Classification accuracy of political blogs based on the graph shift regularization (2.37) and the graph Laplacian regularization (2.38). Figure from [9].

2.4.2. Data Classification

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathbf{A})$ with N vertices that represent N data elements. We assume that two elements are similar to each other if the corresponding vertices are connected; if their connection is directed, the similarity is assumed only in the direction of the edge. We define a signal $\mathbf{s}^{\text{known}}$ on this graph that captures known labels. For a two-class problem, this signal is defined as

$$s_i^{(\text{known})} = \begin{cases} +1, & i\text{th element belongs to class 1,} \\ -1, & i\text{th element belongs to class 2,} \\ 0, & \text{class is unknown.} \end{cases} \tag{2.35}$$

The predicted labels for all data elements are found as the signal that varies the least on the graph $\mathcal{G}$. That it, we find the predicted labels $\mathbf{s}^{(\text{predicted})}$ as the solution to the optimization problem

$$\begin{aligned} &\text{minimize}_{\mathbf{s}\in\mathbb{R}^N} && \|\mathbf{s} - \mathbf{A}_n\mathbf{s}\|_2^2 \\ &\text{subject to} && \|\mathbf{C}(\mathbf{s}^{(\text{known})} - \mathbf{s})\|_2^2 \leq \epsilon \end{aligned} \tag{2.36}$$

where $\mathbf{A}_n$ is the normalized adjacency matrix and $\mathbf{C}$ is an $N \times N$ diagonal matrix such that

$$C_{i,i} = \begin{cases} 1, & \text{if } s_i^{\text{known}} \neq 0, \\ 0, & \text{otherwise.} \end{cases}$$

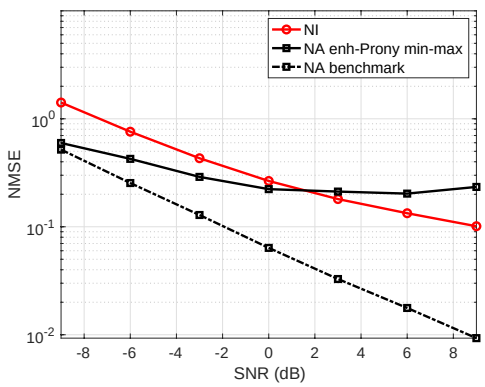
The parameter ϵ controls how well the known labels are preserved. Alternatively, the problem (2.36) can be formulated and solved as

$$\mathbf{s}^{(\text{predicted})} = \underset{\mathbf{s}\in\mathbb{R}^N}{\text{argmin}} \, \|\mathbf{s} - \mathbf{A}_n\mathbf{s}\|_2^2 + \omega_0 \|\mathbf{C}(\mathbf{s}^{(\text{known})} - \mathbf{s})\|_2^2. \tag{2.37}$$

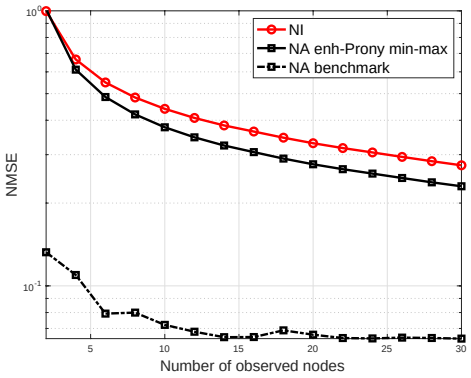
Here, the parameter ω_0 controls the relative importance of the condition in (2.36). Once the predicted signal $\mathbf{s}^{(\text{predicted})}$ is calculated, the unlabeled data elements are assigned to class 1 if $s_i^{\text{predicted}} > 0$ and another class otherwise.

Blog classification [9]. Here, we consider a dataset of 1224 online political blogs [1] introduced in [9]. The blogs contain two classes, conservative and liberal. The dataset is represented by a directed graph with vertices corresponding to blogs and directed edges corresponding to hyperlink references between blogs. We consider the unweighted graph. For comparison, the graph Laplacian based regularization is considered

$$\mathbf{s}^{(\text{predicted})} = \underset{\mathbf{s}\in\mathbb{R}^N}{\text{argmin}} \, \mathbf{s}^\top \mathbf{L}\mathbf{s} + \omega_0 \|\mathbf{C}(\mathbf{s}^{(\text{known})} - \mathbf{s})\|_2^2, \tag{2.38}$$



(a) Denoising.



(b) Interpolation when SNR = 0dB.

Figure 2.8: NMSE performance of different graph signal reconstruction methods for denoising and interpolation task on Molene dataset.

where a variant regularizer based on the normalized graph Laplacian is altered. Note that for (2.38), the graph is set to be undirected by making all edges undirected.

In Figure 2.7, classification accuracy is reported for different ratios of known labels and regularization weight is selected between 1 to 100 for the best performance. The graph shift-based regularization (2.37) significantly outperforms the graph Laplacian regularizer. We also see that for the graph shift regularization (2.37), the directed and the undirected graph give a similar performance when the ratio of the known labels is not very small.

2.4.3. Temperature denoising

Lastly, we apply the graph signal reconstruction technique introduced in Section 2.3 to denoise and interpolate the temperature data. The idea is to view the noisy temperature data collected from different locations as a graph signal sitting on the observation station network. Then, we use the popular graph Laplacian regularization [cf. (2.17) and (2.19)], which we also refer to as the node-invariant regularizer, and the node-adaptive regularization [cf. (2.23) and (2.25)] to reconstruct the signals.

Dataset. We consider two datasets, the Molene[1] and the NOAA[2] weather datasets. Two datasets are both weather temperature data, but the former is collected from a small geographic area while the latter collected from a much larger one. This difference determines if the global smoothness assumption on two datasets is valid.

Molene dataset. The Molene dataset comprises $T = 744$ hourly temperature measurements collected in January 2014 from $N = 32$ weather stations in the region of Brest, France. We treat each weather station as a node of a graph and build a geometric distance graph where each node is connected to its five nearest neighbours. The weight of edge (i, j) is set as $A_{ij} = \exp(-5d^2(i, j))$, where $d(i, j)$ is the Euclidean distance between stations i and j. After removing the mean temperature across space and time, we can view every temporal snapshot as a graph signal.

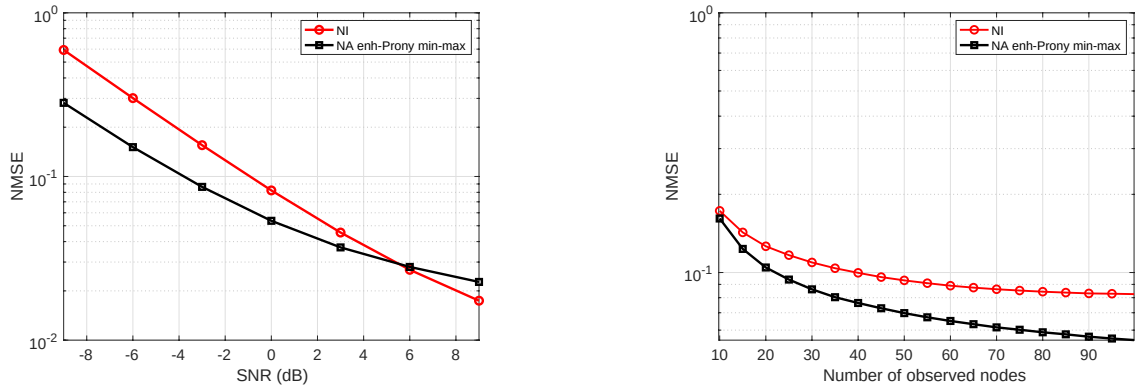
NOAA dataset. The NOAA dataset comprises $T = 8759$ hourly temperature measurements collected across the continental U.S. from $N = 109$ weather stations in 2010. We treat each station as a node of a seven nearest neighbor graph based on geographical distances.

The main difference between two datasets is that NOAA dataset constructs a larger graph compared to Molene dataset in the sense that the graph signal on the graph from NOAA dataset is not reasonably globally smooth, since the data are collected from a large area. This difference allows node-adaptive regularization to work much better in NOAA dataset.

Experiments. We consider the denoising and interpolation tasks on both datasets. For denoising, we evaluate the performance for different signal-to-noise ratios (SNRs) and we artificially add the Gaussian noise to obtain the SNRs ranging in $\{-9\,\text{dB}, -6\,\text{dB}, \ldots, 6\,\text{dB}, 9\,\text{dB}\}$. For interpolation, we collect the noisy observations with a 0 dB SNR from a subset of nodes which contains $\{2, 4, 6, \ldots, 28, 30\}$ nodes for the Molene dataset, and

[1] Raw data available at https://donneespubliques.meteofrance.fr/donneeslibres/Hackathon/ RADOMEH.tar.gz

[2] Raw data available at https://www.ncdc.noaa.gov/data- access/land- based- station- data/ land- based- datasets/climate- normals/1981-2010- normals- data



(a) Denoising.

(b) Interpolation when SNR = 0dB.

Figure 2.9: NMSE performance of different graph signal reconstruction methods for denoising and interpolation task on NOAA dataset.

$\{10, 15, 20, \ldots, 95, 100\}$ nodes for the NOAA dataset. More details on the experiments can be found in [12].
Molene dataset. For the Molene dataset in Figure 2.8, we see that in low SNR situations, the node-adaptive regularization works better than the node-invariant one. This is because the global smoothness assumption becomes less reasonable when the noise level increases. In the interpolation task, the node-adaptive regularization can achieve a smaller NMSE when SNR is 0dB. This is because the node-adaptive regularization captures the local smoothness to recover the signal better.
NOAA dataset. For the NOAA dataset, we would expect a better performance to be obtained based on the node-adaptive regularization, because the temperature data show stronger local smoothness. From Figure 2.9, we see that in both denoising and interpolation tasks, the node-adaptive regularization works better and the performance gap becomes larger compared to the Molene dataset.

2.5. Summary and Further Reading

In this chapter, we studied the signal-graph coupling. First, we introduced the graph shifting operation for signals defined on graphs. Then, we discussed the variation measures of graph signals for both undirected and undirected graphs. These measures can be used as the prior information to reconstruct the graph signal, as we discussed in Section 2.3. Moreover, we introduced a node-adaptive regularizer with an enhanced degree of freedom which can capture the local signal variation measure, leading to an improved performance. Finally, we reviewed three applications where graph signal processing can be used, namely, rating prediction for recommender systems, data classification and sensor data denoising and interpolation.

Bibliography

[1] Lada A Adamic and Natalie Glance. "The political blogosphere and the 2004 US election: divided they blog". In: *Proceedings of the 3rd international workshop on Link discovery*. 2005, pp. 36–43.

[2] Yuanchao Bai et al. "Fast graph sampling set selection using Gershgorin disc alignment". In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 2419–2434.

[3] Pin-Yu Chen and Sijia Liu. "Bias-variance tradeoff of graph laplacian regularizer". In: *IEEE Signal Processing Letters* 24.8 (2017), pp. 1118–1122.

[4] Weiyu Huang, Antonio G Marques, and Alejandro R Ribeiro. "Rating prediction via graph signal processing". In: *IEEE Transactions on Signal Processing* 66.19 (2018), pp. 5066–5081.

[5] Antonio Ortega et al. "Graph signal processing: Overview, challenges, and applications". In: *Proceedings of the IEEE* 106.5 (2018), pp. 808–828.

[6] Paul Resnick et al. "Grouplens: An open architecture for collaborative filtering of netnews". In: *Proceedings of the 1994 ACM conference on Computer supported cooperative work*. 1994, pp. 175–186.

[7] Daniel Romero, Meng Ma, and Georgios B Giannakis. "Kernel-based reconstruction of graph signals". In: *IEEE Transactions on Signal Processing* 65.3 (2016), pp. 764–778.

[8] Aliaksei Sandryhaila and José MF Moura. "Discrete signal processing on graphs". In: *IEEE transactions on signal processing* 61.7 (2013), pp. 1644–1656.

[9] Aliaksei Sandryhaila and Jose MF Moura. "Discrete signal processing on graphs: Frequency analysis". In: *IEEE Transactions on Signal Processing* 62.12 (2014), pp. 3042–3054.

[10] David I Shuman et al. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains". In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[11] Yu-Xiang Wang et al. "Trend filtering on graphs". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 3651–3691.

[12] M. Yang et al. "Node-Adaptive Regularization for Graph Signal Reconstruction". In: *IEEE Open Journal of Signal Processing* (2021), pp. 1–1. DOI: 10.1109/OJSP.2021.3056897.

3

Convolutions Over Graphs

This chapter introduces graph convolutions. First, this chapter makes the relation between time convolution and graph convolution. Then, it defines the concept of graph convolutions and highlights their key properties. Finally, it discusses two practical applications to emphasize the usage of graph convolutions.

3.1. Graph Convolutional Filters

3.1.1. Time Convolution as Graph Operator

In the time domain, the discrete convolution between the input signal $\mathbf{x}$ and the system impulse response $\mathbf{h}$ is denoted as

$$y(n) = \sum_{k=-0}^{K} h_k x_{n-k}, \tag{3.1}$$

where $\mathbf{h} = [h_0, \ldots, h_K]^\top$ denotes the filter parameters and K the filter order. The convolution at time n weighs and combines the K time-shifted (delayed) signals $x_{n-1}, \ldots, x_{n-K}$ with the filter weights $[h_0, \ldots, h_K]$. A related concept, the Z transform of $\mathbf{y}$, $Y(z)$ is obtained as

$$Y(z) = \Big(\sum_{k=0}^{K} h(k) z^{-k} \Big) X(z), \tag{3.2}$$

where z denotes the complex frequency. We will use the term z^{-1} to denote a delay in unit time. The convolution operation is illustrated as a block diagram in Figure 3.1.

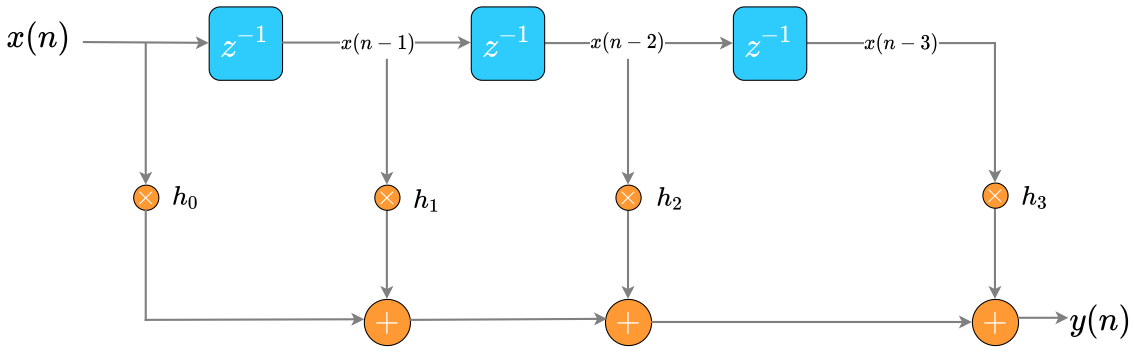


Figure 3.1: Time convolution illustrated with a block diagram: $x(n)$ denotes the signal at time n; block z^{-1} denotes a shift in time by one unit; h_k denotes the filter tap which multiplies with the signal which passes through it; the adder adds signals. The output $y(n)$ is the result of the convolution [cf. (3.1)].

Each node of the graph corresponds to a time index n and a directed edge points to the next time index $n-11$. Figure 2 shows this visualization for a time signal $\mathbf{x}$ of length N with period N. The shift operator $\mathbf{S}$ of

such a graph is of the form

$$\mathbf{S} = \begin{pmatrix} & \vdots & \vdots & \vdots & \\ \dots & 0 & 0 & 0 & \dots \\ \dots & 1 & 0 & 0 & \dots \\ \dots & 0 & 1 & 0 & \dots \\ \dots & 0 & 0 & 1 & \dots \\ & \vdots & \vdots & \vdots & \end{pmatrix}. \tag{3.3}$$

Each element of $\mathbf{x}$ corresponds to an instant in time. Consider an example of four nodes and let the signal be $\mathbf{x} = [1, -1, 0.7, 1.5]^\top$. The matrix $\mathbf{S}$ when applied on $\mathbf{x}$ also shifts it in time as

$$\mathbf{S}\mathbf{x} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} [1, -1, 0.7, 1.5]^\top = [1.5, 1, -1, 0.7]^\top \tag{3.4}$$

The output at time instant i after applying the shift operator is $[\mathbf{S}\mathbf{x}]_i = x_{i-1}$ for $i \geq 1$, i.e., the output at the previous time instant. Similarly, $\mathbf{S}^2$ shifts the signal in time by two instants, $\mathbf{S}^3$ by three instances and so on. We illustrate how $\mathbf{S}$ and its higher powers operates on $\mathbf{x}$ to visualize the time evolution over the graph. The shift operator captures the cyclic shift in time. Figure 3.2 illustrates this relation between a time signal with a periodicity of 4 over the directed cyclic graph of 4 nodes. With each shift using the graph shift operator, we see how the signals shifts over time. 20mm

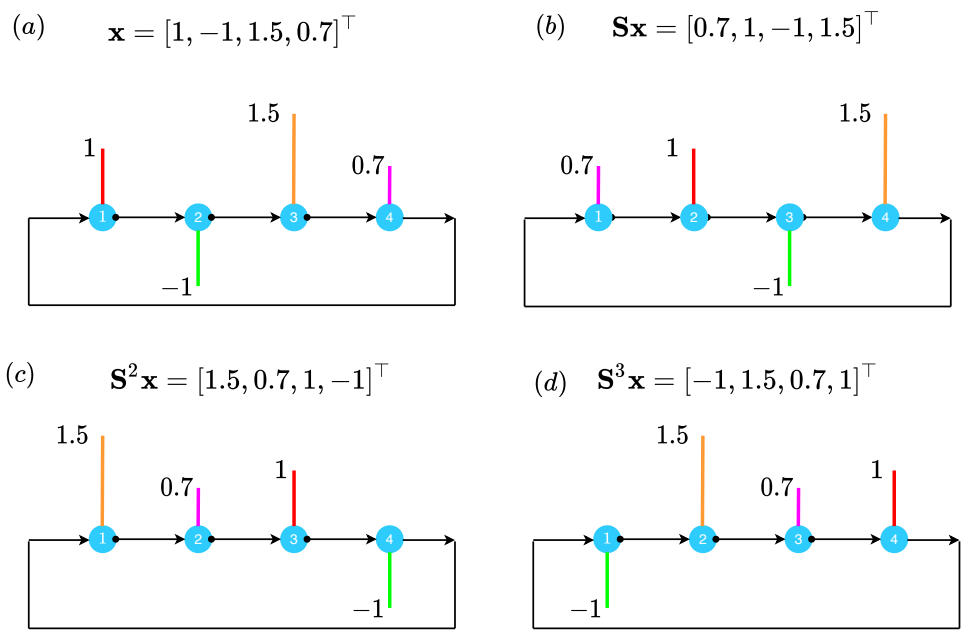


Figure 3.2: Figure shows a periodic signal of period $N = 4$ defined over a cyclic directed graph with 4 nodes; $\mathbf{x}$ denotes the graph signal. The evolution of the time signal is also depicted in terms of successive multiplications with the shift matrix $\mathbf{S}$ of the graph. This captures the relation between the shift operator of the graph and time shifts.

3.1.2. Graph Convolutions

We now formalize the notion of convolutions for arbitrary graphs. Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with node set $\mathcal{V}$ and edge set $\mathcal{E}$. There are $|\mathcal{V}| = N$ nodes and $|\mathcal{E}|$ edges. The matrix $\mathbf{S}$ is the graph shift operator (GSO). The neighborhood of node i, $\mathcal{N}_i$ is the set of all nodes j such that $S_{ij} \neq 0$. The convolution output $\mathbf{y}$ of a signal $\mathbf{x}$ over $\mathbf{S}$ is defined as

$$\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x}, \tag{3.5}$$

where $\sum_{k=0}^{K} h_k \mathbf{S}^k$ is the graph convolutional filter. The convolution in (3.5) consists of a weighted sum of K-shifted versions of $\mathbf{x}$; i.e., $[\mathbf{S}^0\mathbf{x}, \mathbf{S}^1\mathbf{x}, \ldots, \mathbf{S}^K\mathbf{x}]$. These shifts are now done over the an arbitrary graph and are weighted by coefficients $h_0, \ldots, h_K$. Operator $\mathbf{H}(\mathbf{S})$ is also called the Finite Impulse Response (FIR) graph filter. Figure 3.3 shows the block diagram of an FIR graph convolution of order K.

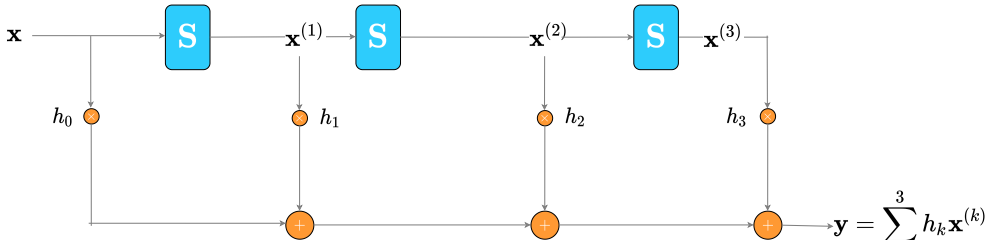


Figure 3.3: Graph convolutional filter of order K illustrated with a block diagram: $\mathbf{x}$ denotes the input graph signal; block $\mathbf{S}$ the shift operator; h_k the filter tap which multiplies the k shifted version $\mathbf{S}^k\mathbf{x}$. The output $y(n)$ is the result of the convolution [cf. (3.1)]..

3.1.3. Locality of Graph Convolutional Filters

The graph convolution comprises a weighted sum of shifted input graph signals. We now focus on the locality of this operation.

- *Order zero shift and one-hop neighborhood*: For $k = 0$, order zero shift $\mathbf{x}^0 = \mathbf{S}^0\mathbf{x} = \mathbf{x}$, which is nothing but the signal itself. For $k = 0$, there are no interactions between the nodes and their neighbors.

- *Order one Shift*: When $k = 1$, the order one shift $\mathbf{x}^{(1)} = \mathbf{S}\mathbf{x}$ is also a graph signal with the value at the ith node given as

$$x_i^1 = [\mathbf{S}\mathbf{x}]_i = \sum_{j=1}^{N} S_{ij} x_j = \sum_{j \in \mathcal{N}_i}^{N} S_{ij} x_j. \tag{3.6}$$

  We can see that the one shift of $\mathbf{x}$ at a node requires information only from the neighbors of that node, also called the one hop neighborhood of i, denoted as $\mathcal{N}_i$.

- *Order two shift and two-hop neighborhood*: For $k = 2$, the second order shift $\mathbf{x}^{(2)} = \mathbf{S}^2\mathbf{x}$ also writes as

$$\mathbf{x}^{(2)} = \mathbf{S}(\mathbf{S}\mathbf{x}) = (\sum_{j=1}^{N} S_{ij}^2 x_j) \tag{3.7}$$

  where $S_{ij}^2 \neq 0$ if and only if node j lies in the 2 hop neighborhood of i, i.e., $j \in \mathcal{N}_i^2 = \bigcup_{k \in \mathcal{N}_i} \mathcal{N}_k$. Hence, the second order shift at each node requires information from all nodes in its 2 hop neighborhood.

- *Order K shift and K-hop neighborhood*: The Kth order shift $\mathbf{x}^{(k)} = \mathbf{S}^K\mathbf{x}$ also writes as

$$\mathbf{x}^{(k)} = \mathbf{S}(\mathbf{S}^{k-1}\mathbf{x}) = (\sum_{j=1}^{N} S_{ij}^k x_j) \tag{3.8}$$

  where $S_{ij}^k \neq 0$ if and only if node j lies in the K hop neighborhood of i, i.e., $j \in \mathcal{N}_i^K = \bigcup_{k \in \mathcal{N}_i^{k-1}} \mathcal{N}_k$. Hence, the Kth order shift at each node requires information from all nodes in its K hop neighborhood.

- *Diffusion Sequence*: For an order $k \geq 2$, we saw that the higher order shifts $\mathbf{x}^{(k)}$. It is obtained by applying the regular shift operator k times on $\mathbf{x}$, i.e.,

$$\mathbf{x}^{(k)} = \mathbf{S}(\mathbf{S}(\ldots)(\mathbf{S}\mathbf{x})) = \mathbf{S}(\mathbf{S}^{k-1}\mathbf{x}) = \mathbf{S}\mathbf{x}^{k-1} \tag{3.9}$$

  where $\mathbf{x}^{k-1}$ is the $k-1$th order shift. For a graph convolution of order K as in (3.5), the set of K shifts of $\mathbf{x}$ denotes the diffusion sequence $\{\mathbf{x}, \mathbf{S}\mathbf{x}, \ldots, \mathbf{S}^K\mathbf{x}\}$.

Figure 3.5 shows how the K-hop neighborhood influences the convolution at a particular node, for $K = 4$.

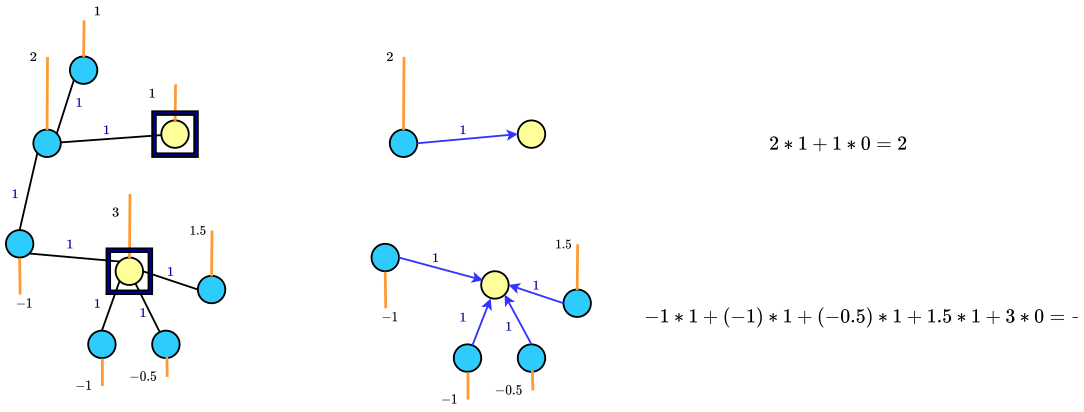


Figure 3.4: Figure comprises three portions: In the first part, a graph is shown along with a graph signal and edge weights (all equal to one. Two nodes are highlighted. In the second part each highlighted node is shown along with its first order neighborhood. The arrows indicate the focus on evaluating the signal at the highlighted nodes. In the third part, the evaluation of $\mathbf{Sx}$ at the highlighted nodes is shown manually.)

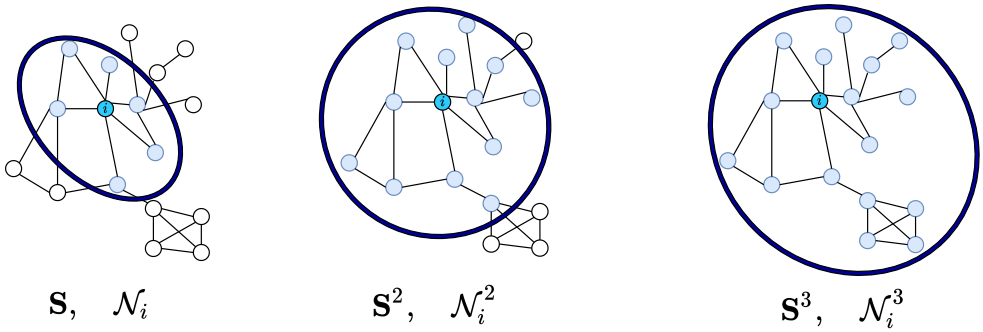


Figure 3.5: Figure shows a graph and the corresponding neighborhoods up to an order of three for a specific node i. Each neighborhood is highlighted by a red circle and all the nodes within it constitute that neighborhood for node i. The corresponding shift matrix is displayed for each neighborhood. We see that with increase in shift order, the different neighbourhoods influence the output at node i.

3.1.4. Properties

In this section, we look at two important properties of graph convolutions, namely that of linear shift invariance and permutation equivariance. Next, we look at how convolutions [cf. (3.1)] are implemented.

- *Linear shift invariance*: For a graph signal $\mathbf{x}$, a k-shift operator $\mathbf{S}^k$ and a graph convolution operator $\mathbf{H}(\mathbf{S})$, the Linear shift invariance translates to

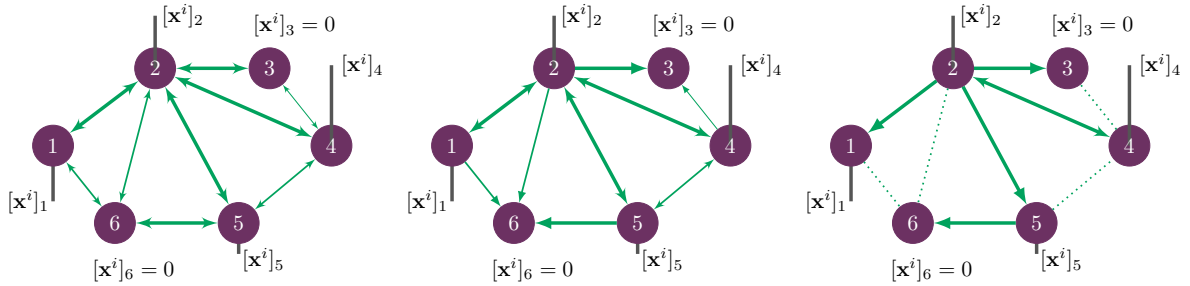
$$\mathbf{H}(\mathbf{S})\mathbf{S}^k = \mathbf{S}^k\mathbf{H}(\mathbf{S}). \tag{3.10}$$

  For (3.10) to hold, $\mathbf{H}(\mathbf{S})$ and $\mathbf{S}$ must share the same eigen-vectors. This condition holds when $\mathbf{H}(\mathbf{S})$ is a polynomial in $\mathbf{S}$. The graph filters defined in (3.5) is a polynomial in $\mathbf{S}$ and is thus shift invariant.

  The linearity of a graph convolutional operator $\mathbf{H}$ states that given two graph signals $\mathbf{x}$ and $\mathbf{y}$ over the same graph, and scalars α, β,

$$\mathbf{H}(\mathbf{S})(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha\mathbf{H}(\mathbf{S})\mathbf{x} + \beta\mathbf{H}(\mathbf{S})\mathbf{y} \tag{3.11}$$

- *Permutation equivariance*: This property states that the graph convolutional filter is invariant to node relabeling. If we relabel the nodes using a permutation operator $\mathbf{P}$, the graph signal $\mathbf{x}$ becomes $\hat{\mathbf{x}} = \mathbf{Px}$ and the shift operator becomes $\hat{\mathbf{S}} = \mathbf{PSP}^\top$. Let $\mathbf{H}(\mathbf{S})$ be the graph convolution prior to relabeling. Permutation equivariance states that if $\mathbf{H}(\hat{\mathbf{S}})$ acts over $\hat{\mathbf{x}}$, the output $\hat{\mathbf{y}}$ is the permuted version of the original output $\mathbf{y}$, where $\mathbf{y} = \mathbf{H}(\mathbf{S})\mathbf{x}$. Graph convolutions are therefore unaffected by a relabelling of the nodes [2].



(a) (b) (c)

Figure 3.6: Building an item-specific graph $\mathbf{B}_i$ from the global correlation graph $\mathbf{B}$. (a) User graph $\mathbf{B}$. Nodes represent users and arrows correlations. Ratings to item i are a graph signal $\mathbf{x}_i$ shown by vertical bars. (b) Treat each undirected edge as two directed edges. Remove any directed edge starting from a user who did not rate item i. (c) Keep the $n = 1$ strongest incoming edge for each user representing the nearest neighbor. The adjacency matrix of this graph is $\mathbf{B}_i$.

- *Recursive and distributed implementation*: We look at the implementation of graph convolutions and discuss its two key properties. First, we look at its recursive nature. Consider successive graph shifts. The kth shift output $\mathbf{x}^{(k)} = \mathbf{S}^k\mathbf{x}$ can be evaluated as $\mathbf{x}^{(k)} = \mathbf{S}(\mathbf{S}^{k-1}\mathbf{x}) = \mathbf{S}\mathbf{z}^{k-1}$. This allows a recursive computation of the successive shifted signals. The complexity of each shift operation $\mathbf{S}\mathbf{x}$ is $\mathcal{O}(|\mathcal{E}|)$, linear in the number of edges. AN order K graph convolution computed recursively has a complexity of $\mathcal{O}(|\mathcal{E}|K)$.

  Second, we highlight the distributed nature of graph convolutions; i.e., the ability of each node to compute its output locally. The scalar x_i^k is the k-shifted signal at the ith node, obtained from $\mathbf{x}^{k-1}$ at $\mathcal{N}_i$. By storing the values $x_i^0, x_i^1, \ldots, x_i^k$ in a vector $\mathbf{z}_i = [x_i^0, x_i^1, \ldots, x_i^k]^\top$ and the coefficients $\mathbf{h}$ locally at node i, the output of the convolution $\mathbf{y}$ [cf. (3.5)] at node i is obtained as

$$y_i = \mathbf{h}^T\mathbf{x}_i, \quad i = 1, \ldots, |\mathcal{V}| \tag{3.12}$$

  which is a local computation. Thus, graph convolutions or FIR graph filters are recursive and distributed.

3.2. Applications

We now look at applications of graph convolutions in recommender systems and node classification.

Consider a recommender system with a set of users $\mathcal{U} = \{1, \ldots, U\}$ and items $\mathcal{I} = \{1, \ldots, I\}$. Let matrix $\mathbf{X}$ of dimension $U \times I$ denote the ratings matrix with X_{ui} the rating provided by user u to item i. If that is not the case, we take X_{ui} to be zero. The objective is to populate matrix $\mathbf{X}$ by exploiting the relationships between users and items contained in the available ratings. We capture these relationships through a graph, which is built following the principles of Nearest Neighbor (NN) collaborative filtering. This graph is used to predict ratings. In user-based NNs, relationships are measured by the Pearson correlation coefficient. Consider a matrix $\mathbf{B} \in \mathbb{R}^{U \times U}$ in which entry B_{uv} measures the correlation between users u and v. Matrix $\mathbf{B}$ is symmetric and can be seen as the adjacency matrix of a user-correlation graph $\mathcal{G}_u = (\mathcal{U}, \mathcal{E}_u)$. The vertex set of $\mathcal{G}_u$ is the user set $\mathcal{U}$ and the edge set $\mathcal{E}_u$ contains an edge $(u, v) \in \mathcal{E}_u$ only if $B_{uv} \neq 0$. Each item i is treated separately and user ratings are collected in vector $\mathbf{x}_i \in \mathbb{R}^U$ (corresponding to the ith column of $\mathbf{X}$). Vector $\mathbf{x}_i$ can be seen as a signal on the vertices of $\mathcal{G}_u$, which uth entry $[\mathbf{x}_i]u = X_{ui}$ is the rating of user u to item i, or zero otherwise; see Figure 3.6. Predicting ratings for item i translates into interpolating the missing values of graph signal $\mathbf{x}_i$. These values are estimated by shifting available ratings to neighboring users. First, we transform the global graph $\mathbf{B}$ to an item-specific graph $\mathbf{B}_i$ which contains only the top-n positively correlated edges per user and normalize their weights; see Figure 3.6. The NN shifted ratings to immediate neighbors can be written as

$$\hat{\mathbf{x}}_i = \mathbf{B}_i\mathbf{x}_i \tag{3.13}$$

which holds true because matrix $\mathbf{B}_i$ respects the sparsity of the user-graph adapted to item i. In item-based NNs, the procedure follows likewise. Matrices $\mathbf{B}$, $\mathbf{B}_i$ can be regarded as instances of a general graph adjacency matrix variable S of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ containing $|\mathcal{V}|$ nodes and $|\mathcal{E}|$ edges. We denote the available rating signal by $\mathbf{x}$ and the estimated rating signal by $\hat{\mathbf{x}}$ so that we write estimators (3.13) with the notation

$$\hat{\mathbf{x}} = \mathbf{S}\mathbf{x} \tag{3.14}$$

Table 3.1: Values of RMSE for the Nearest neighbor graph for the Movielens-100k and Douban data sets. The standard deviation is in the brackets..

	User NN	Item NN
Movielens-100k	0.96 (0.58)	0.96 (0.62)
Douban	0.76 (0.60)	0.8 (0.61)

As it follows from (3.14), NN estimators rely only on ratings present in the immediate surrounding of a node. But higher-order neighbors carry information that can improve prediction and their information should be accounted for accordingly to avoid destructive interference.

Nearest Neighbor Graph Convolutional Filters Estimator (3.14) accounts for the immediate neighbors to predict ratings. Similarly, we can account for the two-hop neighbors via the second-order shift $\mathbf{S}^2\mathbf{x}$. Writing $\mathbf{S}^2\mathbf{x} = \mathbf{S}(\mathbf{S}\mathbf{x})$ shows the second-order shift builds a NN estimator $\mathbf{S}()$ on the previous one $\mathbf{S}\mathbf{x}$. We can also consider neighbors up to K-hops away as $\mathbf{S}^K\mathbf{x} = \mathbf{S}(\mathbf{S}^{K-1}\mathbf{x})$. To balance the information coming from the different resolutions $\mathbf{S}\mathbf{x}; \mathbf{S}^2\mathbf{x}; \ldots; \mathbf{S}^K\mathbf{x}$, we consider a set of parameters $\mathbf{h} = [h_0; \ldots; h_K]^\top$ and build the Kth order NN predictor

$$\hat{\mathbf{x}} = \sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x} = \mathbf{H}(\mathbf{S})\mathbf{x} \tag{3.15}$$

where $\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{K} h_k \mathbf{S}^k$ is referred to as a graph convolutional filter of order K. The ratings $\hat{\mathbf{x}}$ in (3.15) are built as a shift-and-sum of the available ratings $\mathbf{x}$. Particularizing $\mathscr{G}$ to $\mathscr{G}_u$, (3.15) becomes a graph convolutional filter estimator over the user NN graph with estimated ratings for item i

$$\hat{\mathbf{x}}_i = \sum_{k=0}^{K} h_k \mathbf{B}_i^k \mathbf{x}_i = \mathbf{H}(\mathbf{B}_i)\mathbf{x}_i. \tag{3.16}$$

The output $\hat{\mathbf{x}}$ in (3.15) amounts to a complexity of order $\mathscr{O}(|\mathscr{E}|K)$.

Problem formulation and solution: Let $\mathscr{R}$ be the set of all observed user-item interactions. Using the definitions of $\mathbf{B}_i$ and $\mathbf{x}_i$ as mentioned above, the rating for i as predicted by the order K convolution for user u is $\hat{X}_{ui} = [\big(\sum_{k=0}^{K} h_k \mathbf{B}_i^k\big)\mathbf{x}_i]_u$. This draws a squared prediction error $(\hat{X}_{ui} - X_{ui})^2$. The total error over set $\mathscr{R}$ is

$$\sum_{(u,i)\in\mathscr{R}} ([\big(\sum_{k=0}^{K} h_k \mathbf{S}_i^k\big)\mathbf{x}_i]_u - X_{ui})^2, \tag{3.17}$$

where $\mathbf{h} = [h_0, \ldots, h_k]^\top$ contains the convolution (prediction) parameters. For training, we take a set $\mathscr{T} \subset \mathscr{R}$ and minimize the prediction error over it w.r.t $\mathbf{h}$ as follows:

$$\underset{\mathbf{h}\in\mathbb{R}^{K+1}}{\text{minimize}} \sum_{(u,i)\in\mathscr{T}} ([\big(\sum_{k=0}^{K} h_k \mathbf{B}_i^k\big)\mathbf{x}_i]_u - X_{ui})^2 + \gamma||\mathbf{h}||_2^2, \tag{3.18}$$

where $\gamma||\mathbf{h}||_2^2$ imposes smoothness over the optimal $\mathbf{h}$, i.e, the elements in $\mathbf{h}$ do not vary much. Problem (3.20) is a regularized least square problem that is convex for $\gamma > 0$ and has an optimal solution.

Results We show some basic results for rating prediction on two data-sets, the Movielens-100k and Douban data set. Table 3.1 shows the Root mean square error (RMSE) for prediction for both user and item collaborative filtering. In each case, the graph of choice is the nearest neighbor graph. Additionally, Figure 3.7 sheds light on the role of the filter order K and the number of nearest neighbors on the RMSE for Movielens 100k. The upper and lower maps are for the user and item graphs respectively. Increase in nearest neighbors reduces the RMSE for $K = 1, 3$ in the user graph and $K = 2$ in the item graph. For item graph, an increase in K degrades the peformance.

3.2.1. Blog Classification

Classification. We here consider the task of classifying online blogs as conservative or liberal by knowing only hyperlinks between them and partial labels [5]. Often, is not possible to read each blog and label it. So, a small subset of blogs are read, labeled, and the remaining blogs are to be labeled using the available information and the graph structure. This is an example of semi-supervised learning on graphs. Figure 3.8 shows a small

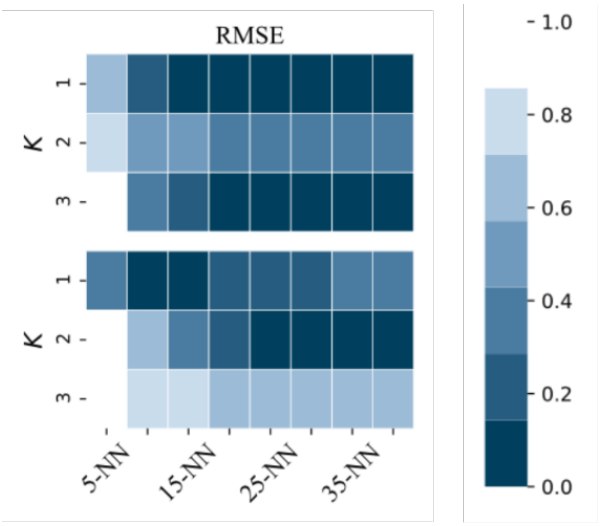


Figure 3.7: Effect of filter order K and the number of nearest neighbors on the RMSE for the Movielens100k data set.

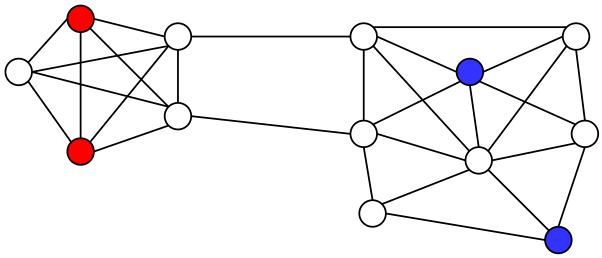


Figure 3.8: Figure shows two well clustered set of nodes, each cluster belonging to a class. The coloured nodes (red and blue) denote the labeled nodes in each class. The uncolored nodes are unlabeled. In semi-supervised learning, the objective is to label the unlabeled nodes using the labeled nodes and the graph structure.

example of such a setup. The blog network graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ has $N = 1224$ nodes which represent online blogs. Let $\mathcal{R}_{trn} = (v, l_v)$ be the training set with each element comprising a labeled node v with its label l_v. Each node v represents a political blog having label $l_v = +1$ for being conservative and $l_v = -1$ for being liberal. Let $\mathbf{x}$ be the graph signal with $x_v = l_v$ if $v \in \mathcal{R}_{trn}$ and zero otherwise. For the purpose of training, unlabeled nodes take a label of zero. Let $\mathcal{R}_{tst}$ be the test set comprising the unlabeled nodes and their true labels. In addition, let $\mathbf{y}_{true}$ be the vector containing the labels of all nodes. The shift operator $\mathbf{S}$ has $S_{ij} = 1$ when blog i is connected to blog j through a hyperlink.

The output $\mathbf{y} = \sum_{p=1}^{P} h_p \mathbf{S}^p \mathbf{x}$ is the diffusion of the labels of the nodes in $\mathcal{R}_{trn}$. Authors in [6] use it to classify the unlabeled nodes as follows: node i is conservative if $y_i > 0$ and liberal if $y_i < 0$. The task now remains to design the parameters h of the filter. Designing the classifier in (**??**) requires a criterion for the classification performance. The criterion is a loss function $l(\mathbf{y}, \mathbf{y}_{true})$ involving the output $\mathbf{y}$ and the true labels $\mathbf{y}_{true}$ evaluated at the nodes in $\mathcal{R}_{trn}$. One example could be the squared loss; i.e.,

$$l(\mathbf{y}, \mathbf{y}_{true}) = \sum_{i \in \mathcal{R}_{trn}} (\mathbf{y}_i - [\mathbf{y}_{true}]_i)_2^2 + g(\mathbf{y}) \tag{3.19}$$

where $g(\cdot)$ is a regularizer on the classifier output. To solve for the filter, we minimize this loss over $\mathbf{h}$ by solving

$$\underset{\mathbf{h} \in \mathbb{R}^{K+1}}{\text{minimize}} \sum_{i \in \mathcal{R}_{trn}} (\mathbf{y}_i - [\mathbf{y}_{true}]_i)_2^2 + g(\mathbf{y}) \tag{3.20}$$

$$\text{subject to} \quad \mathbf{y} = \sum_{k=0}^{K} h_k \mathbf{S}_i^k \mathbf{x} \tag{3.21}$$

Depending on the choice of $g(\cdot)$, we obtain different classifiers.

3.3. Summary and Further Reading

Readers interested to learn more about the concepts mentioned in this chapter are encouraged to look up the following works: For graph convolutions, refer to [5, 7]. For distributed implementation, refer to [7] and for applications of distributed filters, refer to [1]. For recommender systems with graph convolutions, refer to [4, 3]. The graph filtering approach to blog classification can be found in more detail in [5]. For permutation equivariance and shift invariance of graph convolutions, refer to [2].

Bibliography

[1] Mario Coutino, Elvin Isufi, and Geert Leus. “Advances in distributed graph filtering”. In: *IEEE Transactions on Signal Processing* 67.9 (2019), pp. 2320–2333.

[2] Fernando Gama, Joan Bruna, and Alejandro Ribeiro. “Stability properties of graph neural networks”. In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 5680–5695.

[3] Xiangnan He et al. “LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation”. In: *arXiv preprint arXiv:2002.02126* (2020).

[4] Weiyu Huang, Antonio G Marques, and Alejandro R Ribeiro. “Rating prediction via graph signal processing”. In: *IEEE Transactions on Signal Processing* 66.19 (2018), pp. 5066–5081.

[5] Aliaksei Sandryhaila and José MF Moura. “Discrete signal processing on graphs”. In: *IEEE transactions on signal processing* 61.7 (2013), pp. 1644–1656.

[6] Aliaksei Sandryhaila and Jose MF Moura. “Discrete signal processing on graphs: Frequency analysis”. In: *IEEE Transactions on Signal Processing* 62.12 (2014), pp. 3042–3054.

[7] Santiago Segarra, Antonio G Marques, and Alejandro Ribeiro. “Optimal graph-filter design and applications to distributed linear network operators”. In: *IEEE Transactions on Signal Processing* 65.15 (2017), pp. 4117–4131.

4

Fourier Analysis on Graphs

In this chapter, we will look at the frequency domain of graphs and graph signals. We start with the discrete Fourier transform for time signals, show its link to graphs and then define the graph Fourier transform (GFT). Next, we derive the GFT for the Laplacian and adjacency matrices and discuss how their eigenvalues and eigenvectors are related to frequency components. Next, we derive the graph filter frequency response and discuss properties like locality and filter order. Finally, we look at two applications which require the graph frequency domain, distributed signal de-noising and compressive spectral clustering.

4.1. Graph Fourier Transform

In this section, we define the graph Fourier transform. We start by drawing connections between the Discrete Fourier Transform (DFT) of time signals and the Fourier transform of graph signals.

4.1.1. From DFT to GFT

Discrete Fourier Transform. A periodic time signal $\mathbf{x} = [x_1, \dots, x_N]^\top$ can be interpreted as a graph signal over a directed cyclic graph of N nodes with the Shift operator $\mathbf{C} \in \{0,1\}^{N \times N}$

$$\mathbf{C} = \begin{pmatrix} 0 & 0 & 0 & \cdots & \cdots & 1 \\ 1 & 0 & 0 & \ddots & & 0 \\ 0 & 1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 & 0 \\ \vdots & & \ddots & 1 & 0 & 0 \\ 0 & \cdots & \cdots & 0 & 1 & 0 \end{pmatrix}. \tag{4.1}$$

The eigendecomposition of a cirulant matrix is special and its kth eigenvector $\mathbf{C} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^H$ is

$$\mathbf{v}_k = [\omega_N^{0k}, \omega_N^{1k}, \omega_N^{2k}, \dots, \omega_N^{(N-1)k}]^\top \tag{4.2}$$

where $\omega_N = e^{\frac{j2\pi}{N}}$ is a complex exponential with frequency $\frac{j2\pi}{N}$.[1] Substituting the values of k in (4.2) and ω_N for $N = 5$, $\mathbf{V}$ is

$$\mathbf{V} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & e^{\frac{j2\pi 1}{5}} & e^{\frac{j2\pi 2*1}{5}} & e^{\frac{j2\pi 3*1}{5}} & e^{\frac{j2\pi 4*1}{5}} \\ 1 & e^{\frac{j2\pi 2}{5}} & e^{\frac{j2\pi 2*2}{5}} & e^{\frac{j2\pi 3*2}{5}} & e^{\frac{j2\pi 4*2}{5}} \\ 1 & e^{\frac{j2\pi 3}{5}} & e^{\frac{j2\pi 2*3}{5}} & e^{\frac{j2\pi 3*3}{5}} & e^{\frac{j2\pi 4*3}{5}} \\ 1 & e^{\frac{j2\pi 4}{5}} & e^{\frac{j2\pi 2*4}{5}} & e^{\frac{j2\pi 3*4}{5}} & e^{\frac{j2\pi 4*4}{5}} \end{pmatrix}. \tag{4.3}$$

[1] A complex exponential is represented as e^{jx}, where j denotes the imaginary number $\sqrt{-1}$. The complex exponential is expressed as $e^{jx} = \cos(x) + j\sin(x)$. The complex exponential represents a sine and sine function on the real and imaginary axis at the same frequency. It has unit magnitude, i.e. $|e^{jx}| = 1$. It also represents a point on the unit circle.

The eigenvalues diag($\mathbf{\Lambda}$) represent the N temporal frequencies. These eigenvalues are the N primitive roots of unity, the kth eigenvalue being $e^{\frac{j2\pi k-1}{N}}$. The eigenvectors $[\mathbf{v}_1,\dots,\mathbf{v}_N]^\top$ are the modes of the temporal graph, corresponding to their temporal frequencies. Each eigenvector is sampled in time from the corresponding temporal frequency (eigenvalue). Upon inspection, $\mathbf{V}$ is the Discrete Fourier Transform matrix for temporal signal $\mathbf{x}$. The DFT of signal $\mathbf{x}$ is thus $\hat{\mathbf{x}} = \mathbf{V}^{-1}\mathbf{x}$. In other words, the DFT of a time signal is its projection onto the eigenvectors of the shift operator of the directed cyclic graph (i.e., the graph that captures the temporal relations of periodic signals).

A bandlimited time signal $x(t)$ is a signal whose Discrete Fourier Transform X_k is non-zero for a subset of the set of frequencies $\mathscr{F} = \{\frac{j2\pi k}{N}\}$ with $k = \{0,\dots,N-1\}$. Mathematically this writes as

$$X_k \neq 0, \quad k \in \mathscr{F}_B \subset \mathscr{F}. \tag{4.4}$$

This means that frequencies greater than ω_B are not contributing to forming $x(t)$, so the maximum amount of oscillations in $x(t)$ is limited.

For an arbitrary graph, the shift operator $\mathbf{S}$ encodes its structure in a matrix form. The shift operator has the eigendecomposition:

$$\mathbf{S} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}. \tag{4.5}$$

Graph modes. The modes of the graph are the eigenvectors $\mathbf{V} = [\mathbf{v}_1,\dots,\mathbf{v}_N]$ of $\mathbf{S}$. Previously, we saw for periodic time signals, the modes are the related to the temporal frequencies. Due to the arbitrary structure of the graph, the modes of the graph are related to the eigenvalues, which carry their own interpretation of frequency.

Graph frequencies. Like the eigenvalues above for the temporal signal denote the temporal frequencies, the eigenvalues $\mathbf{\Lambda}$ of (38) will denote the graph frequencies.

Graph Fourier Transform. The graph Fourier transform of a graph signal $\mathbf{x}$ is defined as

$$\hat{\mathbf{x}} = \mathbf{V}^{-1}\mathbf{x}. \tag{4.6}$$

Element $\hat{x}_i$ denotes the ith frequency component of $\mathbf{x}$, i.e., how much it is related to the frequency denoted by the ith eigenvalue of $\mathbf{S}$.

Bandlimited graph signal. A bandlimited graph signal $\mathbf{x}$ has $\hat{x}_i = 0$ for $i \in \mathscr{I} \subset \{1,\dots,N\}$. This means it is restricted only to a set of available graph frequencies.

While the definition of the GFT is similar for both directed and undirected graph, its spectral interpretation changes. In the sequel we focus on detailing the latter by using the Laplcian for undirected graphs [11] and the adjacency matrix for directed graphs [7].

4.1.2. GFT w.r.t the Laplacian

For an undirected graph, the graph Laplacian matrix $\mathbf{L} = \text{diag}(\mathbf{D}) - \mathbf{W}$ is symmetric and has the eigendecomposition

$$\mathbf{L} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^H \tag{4.7}$$

with orthonormal eigenvectors $\mathbf{V} = [\mathbf{v}_1,\dots,\mathbf{v}_N]$ and eigenvalues $\mathbf{\Lambda} = \text{diag}(\lambda_1,\dots,_N)$. Since $\mathbf{L}$ is positive semi-definite, its eigenvalues are non-negative scalars and can be ordered as $0 = \lambda_1 \leq \lambda_2 \leq \lambda_3 \dots \leq \lambda_N = \lambda_{max}$. As we shall derive in the sequel, these ordered eigenvalues will play the role of the graph frequencies for undirected graphs, where $\lambda_1 = 0$ will be the DC component and $\lambda_N = \lambda_{max}$ will be the maximum graph frequency.

For a graph signal $\mathbf{x}$, its smoothness w.r.t the graph Laplacian is $S_2(\mathbf{x}) = \mathbf{x}^\top\mathbf{L}\mathbf{x}$. When the signal is the ith eigenvector, i.e., $\mathbf{x} = \mathbf{v}_i$, the smoothness becomes $S_2(\mathbf{v}_i) = \mathbf{v}_i^\top\mathbf{L}\mathbf{v}_i = \lambda_i$, where we use the properties $\mathbf{L}\mathbf{v}_i = \lambda_i\mathbf{v}_i$ and $||\mathbf{v}_i||_2^2 = 1$. That is the smoothness variation for the ith eigenvector corresponds to the ith graph frequency (eigenvalue) λ_i. The eigenvalue λ_0 can be obtained by minimizing the Rayleigh quotient problem

$$\begin{aligned} \lambda_0 = \min_{\mathbf{x}\in\mathbb{R}^N} \quad & \mathbf{x}^\top\mathbf{L}\mathbf{x} \\ \text{subject to} \quad & ||\mathbf{x}||_2 = 1 \end{aligned} \tag{4.8}$$

where the optimum will be the zero-th eigenvalue. The solution for problem (4.8) corresponds to $\lambda_1 = 0$ and $\mathbf{v}_1 = \frac{1}{\sqrt{N}}\mathbf{1}$. Thus, eigenvalue $\lambda_1 = 0$ is the one for which the corresponding eigenvector has the minimum variation, i.e., it is constant. The remaining eigenvalues $\lambda_2,\dots,\lambda_N$ can be solved iteratively using the orthonormal

property of the eigenvectors

$$\begin{aligned} \lambda_l = \min_{\mathbf{x}\in\mathbb{R}^N} \quad & \mathbf{x}^\top \mathbf{L}\mathbf{x} \\ \text{subject to} \quad & ||\mathbf{x}||_2 = 1 \\ & \mathbf{x} \perp \text{span}\{\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{l-1}\} \quad l = 1, \ldots, N-1. \end{aligned} \tag{4.9}$$

Solving problem (4.9) sequentially gives us the eigenvalues of $\mathbf{L}$ and the corresponding eigenvectors in ascending order. Since the eigenvalues are obtained from the Laplacian quadratic form of the corresponing eigenvetor, a smaller eigenvalue means its eigenvector varies less over the graph. Thus, $\lambda_1 = 0$ corresponds to the eigenvector $\mathbf{v}_1 = \frac{1}{\sqrt{N}}\mathbf{1}$ with no variation over the graph. This mode is equivalent to the DC mode in the Fourier expansion of time signals. For higher values of λ_i, the corresponding $\mathbf{v}_i$ will exhibit higher variation over the graph. Figure 4.1 illustrates the constant eigenvector along with an intermediate and high-varying eigenvector over the graph along with their respective eigenvalues.

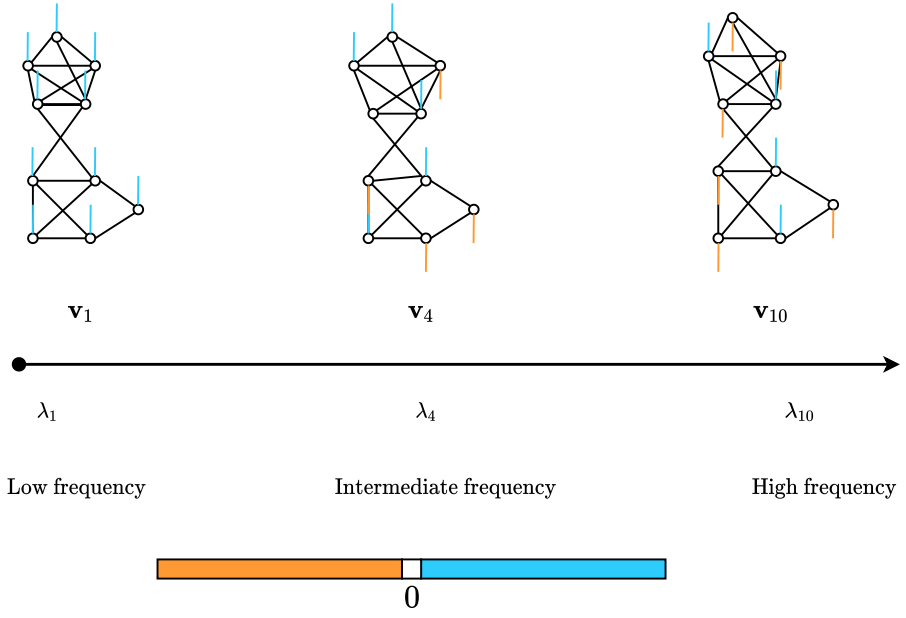


Figure 4.1: Modes of the graph Laplacian expressed as signals for a particular graph of 10 nodes, with increase in graph frequency. The eigenvectors $\mathbf{v}_0$, $\mathbf{v}_4$ and $\mathbf{v}_{10}$ are the low, intermediate and high frequency modes corresponding to λ_0, λ_4 and λ_{10}, respectively.

A low pass graph signal $\mathbf{x}$ has a low smoothness $S_2(\mathbf{x})$. It can be expressed as a linear combination of the first few modes of $\mathbf{L}$; i.e., $\mathbf{x} = \mathbf{V}_K\mathbf{h}$ where $\mathbf{V}_K$ has the first $K \leq N$ columns of $\mathbf{V}$ and $\mathbf{h} = [h_1, \ldots, h_K]$ contains the combining coefficients. A high pass graph signal has a high value of smoothness relative to the Laplacian. It can be expressed as a linear combination of the last few modes of $\mathbf{L}$; i.e., $\mathbf{x} = \mathbf{V}_{N-K}\mathbf{h}$ where $\mathbf{V}_K$ has the last $K \leq N$ columns of $\mathbf{V}$ and $\mathbf{h} = [h_1, \ldots, h_K]$ contains the combining coefficients. Since each such mode has high variability, so will their linear combination.

Note that this analysis can also be conducted in a similar way also for the normalized Laplacian and for any Laplacian-like matrix; but this one is mostly preferred in the literature.

4.1.3. GFT w.r.t the Adjacency Matrix

In this section, we derive the GFT w.r.t. the Adjacency matrix as a shift operator. We will also observe how the graph frequencies the graph frequencies are ordered on the basis of their frequency of variation. We use the normalized adjacency matrix in this section [7], given as

$$\mathbf{A}_n = \frac{1}{\lambda_{max}}\mathbf{A} \tag{4.10}$$

where λ_{max} is the largest eigenvalue of $\mathbf{A}$. To proceed, we define the total variation (TV) of a graph signal $\mathbf{x}$ w.r.t. $\mathbf{A}_n$ as

$$\text{TV}(\mathbf{x}) = ||\mathbf{x} - \mathbf{A}_n\mathbf{x}||_1 \tag{4.11}$$

which is the sum over nodes of the absolute difference between the graph signal $\mathbf{x}$ and the shifted version $\mathbf{A}_n\mathbf{x}$. The higher the TV, the more signal $\mathbf{x}$ and changes from its shifted version. For $\mathbf{A}$ being diagonalizable ($\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$) and for an eigenvalue-eigenvector pair $(\lambda, \mathbf{v})$, the total variation is

$$TV(\mathbf{v}) = |1 - \frac{\lambda}{|\lambda_{max}|}|||\mathbf{v}||_1. \tag{4.12}$$

So, viewing now each eigenvector $\mathbf{v}$ as a graph signal, the total variation is $TV_{\mathcal{G}}(\lambda) = |1 - \frac{\lambda}{|\lambda_{max}|}|$.

Real eigenvalues When $\mathbf{A}$ is symmetric matrix, typical of undirected graphs, its eigenvalues are real. Consider two distinct real eigenvalues λ_n and λ_m of a symmetric $\mathbf{A}$ with eigenvectors $\mathbf{v}_n$ and $\mathbf{v}_m$ respectively, both of them normalized to have unit 1-norm. Additionally, consider $\lambda_n > \lambda_m$. The TV of $\mathbf{v}_n$ will be lesser than that of $\mathbf{v}_m$. Thus, for symmetric Adjacency matrix, if its real eigenvalues can be ordered as $\lambda_1 < \lambda_2 < \ldots < \lambda_N$, then the total variation orders as $\text{TV}(\mathbf{v}_1) > \text{TV}(\mathbf{v}_2) > \ldots > \text{TV}(\mathbf{v}_N)$, i.e., the smaller the eigenvalue, the greater the notion of frequency and the higher will be the variation of its eigenvector. Figure illustrates this frequency ordering. Accordingly, a low-pass graph signal for the adjacency matrix writes as a linear combination of the eigenvectors corresponding to the largest eigenvalues of $\mathbf{A}$, whose modes exhibit low total variation. A high-pass graph signal is a combination of the modes corresponding to the lowest eigenvalues of $\mathbf{A}$, which exhibit high total variation.

Complex eigenvalues When $\mathbf{A}$ is not a symmetric matrix, typical of directed graphs, its eigenvalues are complex. Consider two distinct complex eigenvalues λ_n and λ_m of a directed $\mathbf{A}$ with eigenvectors $\mathbf{v}_n$ and $\mathbf{v}_m$ respectively, both of them normalized to have unit 1-norm. If λ_n is closer to $|\lambda_{max}|$ in the complex plane than λ_m, i.e., $|\lambda_n - |\lambda_{max}|| < |\lambda_m - |\lambda_{max}||$, then the total variation $\text{TV}(\mathbf{v}_m)$ is greater than $\text{TV}(\mathbf{v}_n)$. Any two eigenvalues which lie on the circumference of a circle drawn with centre at the point $(|\lambda_{max}|, 0)$ will exhibit the same total variation. This ordering is illustrated in Figure.

Comparison with laplacian spectrum The main difference between the Adjacency and Laplacian spectrum is w.r.t the ordering of the eigenvalues. For the Laplacian, the eigenvalues will always be real whereas for the Adjacency matrix, the eigenvalues may be real or complex. For the Laplacian, a lower eigenvalue will be associated with a lower frequency of variation, whereas for a symmetric adjacency matrix, a lower eigenvalue will be associated with a higher frequency of variation. The notion of frequency ordering in both cases are opposed to each other. In addition, for directed graphs the ordering is done not on the basis of the eigenvalues but on their distance from the absolute of the maximum eigenvalue. Figure 4.2 illustrates the ordering for the Laplacian, a symmetric and directed adjacency matrix. The ordering follows the rules described above.

For directed graphs research is still going especially when $\mathbf{A}$ is not diagonalisable, with notable works being [8, 9].

4.2. Frequency Analysis of Graph Filters

Given the Fourier transform for graph signals, we can formally derive the frequency response of a graph filter. Upon defining this frequency response, we will define low, high and band-pass graph filters. Lastly, we look at some properties of graph filters.

4.2.1. Graph Filter Frequency Response

Recall that the graph convolutional filter over graph $\mathcal{G}$ with shift operator $\mathbf{S}$ is

$$\mathbf{y} = \sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x} \tag{4.13}$$

with $\mathbf{h} = \{h_k\}_{k=0}^{K}$ being the filters coefficients, $\mathbf{x}$ the input signal and $\mathbf{y}$ the output signal. Using the eigen-decomposition of $\mathbf{S}$ as $\mathbf{S} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ and the identity $\mathbf{S}^k = \mathbf{V}\mathbf{\Lambda}^k\mathbf{V}^{-1}$, we can write (4.13) as

$$\mathbf{y} = \sum_{k=0}^{K} h_k \mathbf{V}\mathbf{\Lambda}^k\mathbf{V}^{-1}\mathbf{x}. \tag{4.14}$$

Multiplying both sides of (4.14) by $\mathbf{V}^{-1}$, we get

$$\mathbf{V}^{-1}\mathbf{y} = \sum_{k=0}^{K} h_k \mathbf{V}^{-1}\mathbf{V}\mathbf{\Lambda}^k\mathbf{V}^{-1}\mathbf{x}. \tag{4.15}$$

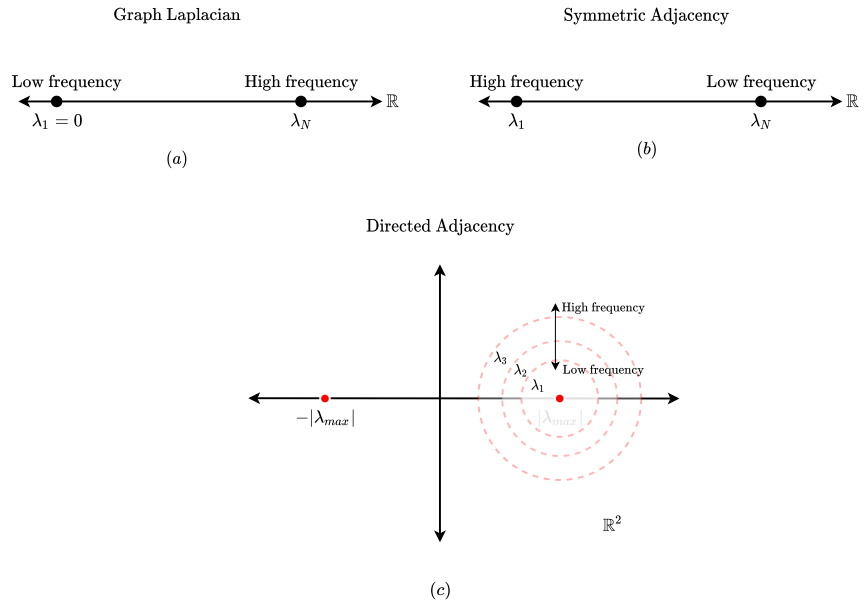


Figure 4.2: Eigenvalue ordering according to frequency. First figure shows the ordering on the real axis for the graph Laplacian; second shows the same for a symmetric adjacency matrix. The final part shows the ordering for a directed adjacency matrix with complex eigenvalues.

Using the definition of GFT, $\hat{\mathbf{y}} = \mathbf{V}^{-1}\mathbf{y}$, $\hat{\mathbf{x}} = \mathbf{V}^{-1}\mathbf{x}$ and he fact that $\mathbf{V}^{-1}\mathbf{V} = \mathbf{I}$, we finally get

$$\hat{\mathbf{y}} = \sum_{k=0}^{K} h_k \mathbf{\Lambda}^k \hat{\mathbf{x}}. \tag{4.16}$$

.

Equation (4.16) gives us filter input-output relation in the frequency domain. The input GFT $\hat{\mathbf{x}}$ and output GFT $\hat{\mathbf{y}}$ are related through the diagonal matrix $\mathbf{H}(\mathbf{\Lambda}) = \sum_{k=0}^{K} h_k \mathbf{\Lambda}^k$. This matrix contains on its main diagonal the frequency response of the filter. The ith output GFT coefficient $\hat{y}_i$ is related to the ith input GFT coefficient $\hat{x}_i$ as

$$\hat{y}_i = \sum_{k=0}^{K} h_k \lambda_i^k \hat{x}_i \tag{4.17}$$

where λ_i is the ith eigenvalue of $\mathbf{S}$. Given the coefficients $\mathbf{h}$, the graph frequency response is a polynomial in the frequency variable λ of the form

$$h(\lambda) = \sum_{k=0}^{K} h_k \lambda^k \tag{4.18}$$

so that (4.17) can be expressed as

$$\hat{y}_i = h(\lambda_i)\hat{x}_i. \tag{4.19}$$

In the matrix form, the graph frequency response is

$$\mathbf{H}(\mathbf{\Lambda}) = \sum_{k=0}^{K} h_k \mathbf{\Lambda}^k \tag{4.20}$$

and (4.16) becomes

$$\hat{\mathbf{y}} = \mathbf{H}(\mathbf{\Lambda})\hat{\mathbf{x}}. \tag{4.21}$$

That is, graph convolutions in the vertex domain become point-wise multiplications between GFTs in the frequency domain, just like convolutions between time signals become pointwise multiplications between their DFTs in the frequency domain. We now define $h(\lambda)$ for three standard graph filters, i.e., the low-pass, high-pass and band-pass graph filter. To make this analysis simple, we will consider the ordering w.r.t. the Laplacian [11] but a similar one can also be used for the adjacency matrix [7].

- *Low-pass filter:* An ideal low-pass graph filter, $h_{lp}(\lambda)$ is defined as

$$\begin{aligned} h_{lp}(\lambda) &= 1, \quad \lambda \leq \lambda_l \\ h_{lp}(\lambda) &= 0, \quad \lambda > \lambda_l \end{aligned}. \tag{4.22}$$

  This filter outputs $\hat{y}_i = 0$ for all graph frequencies greater than a cut-off frequency λ_l. This frequency response removes high frequency content and retains the low frequency content up to λ_l.

- *High-pass filter:* An ideal high-pass graph filter, $h_{hp}(\lambda)$ is defined as

$$\begin{aligned} h_{hp}(\lambda) &= 0, \quad \lambda \leq \lambda_h \\ h_{hp}(\lambda) &= 1, \quad \lambda > \lambda_h \end{aligned}. \tag{4.23}$$

  This filter outputs $\hat{y}_i = 0$ for all graph frequencies lesser than a cut-off frequency λ_h. This frequency response removes low frequency content and retains the high frequency content from λ_h onwards.

- *Band-pass filter:* An ideal high-pass graph filter, $h_{bp}(\lambda)$ is defined as

$$\begin{aligned} h_{bp}(\lambda) &= 0, \quad \lambda < \lambda_l \\ h_{bp}(\lambda) &= 1, \quad \lambda_l \leq \lambda \leq \lambda_h. \\ h_{bp}(\lambda) &= 0, \quad \lambda > \lambda_h \end{aligned} \tag{4.24}$$

  This filter outputs $\mathbf{y}_i = 0$ for graph frequencies lower than λ_l and higher than λ_h. This frequency response retains all content between the frequencies λ_l and λ_h and removes everything else. Figure 4.3 illustrates h_{lp}, h_{hp} and h_{bp}.

To define these filter responses for the adjacency matrix, we take into notion the effect of frequency ordering of its eigenvalues. Thus, the ideal low pass filter response will resemble the ideal high pass response for the Laplacian and the ideal high pass response will resemble the ideal low pass response for the Laplacian. The band-limited filter response will be similar.

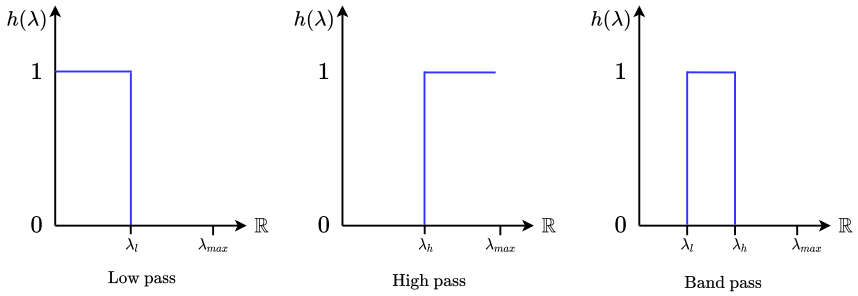


Figure 4.3: The ideal low-pass, high-pass and band-pass frequency responses.

4.2.2. Notes on the Frequency Response

- *Universal filter response*: Graph filters are defined by their coefficients **h**. Given a set of coefficients, their frequency response $h()$ [cf.(4.18)] can be seen as an analytic polynomial function in a variable λ. The specific graph over which this filter will be implemented will instantiate the shift operator **S**, hence the eigenvalues λ. So, for an arbitrary variable λ, we say that the frequency response $h(\lambda)$ is a universal frequency response independent of the graph and it is a continuous function in λ. The filter responses [cf. (4.18),(4.27),(4.23),(4.24)] are examples of universal filter responses. Figure 4.4 shows a particular $h(\lambda)$.

- *Graph-specific filter response*: When the graph is fixed, hence also the shift operator **S** and eigenvalues, we have the universal filter frequency response evaluated at a finite set of graph frequencies $\lambda_1, ..., \lambda_N$. In this case we will talk about the graph-specific frequency response. This response is discrete, with the ith frequency response $h(\lambda_i)$ being the universal response $h(\lambda)$ evaluated at eigenvalue λ_i. Figure 4.4 shows the graph-specific frequency response.

- *Illustration*: Figure 4.4 illustrates both the universal and graph-specific filter response. The blue line indicates the universal response, a polynomial in λ. The graph-specific response is shown in red, along with its spectrum. We see that the graph-specific response is the universal response sampled at its spectrum.

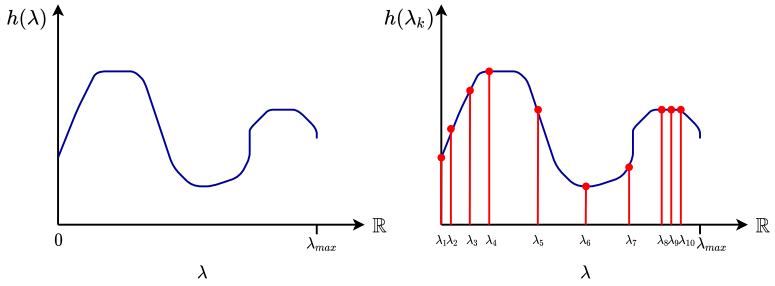


Figure 4.4: Universal and graph-specific frequency response. The first plot in blue shows a polynomial frequency response $h(\lambda)$ independent of any graph. The second plot shows the same response evaluated at 10 discrete frequencies, which is the spectra of a graph with $N = 10$ nodes.

4.2.3. Spectral Locality of Graph Filters

A particular parameter in graph filters is the filter order K. This order defines the order of the convolution and as it can be see from its frequency response in (4.18), it defines also the ability of the filter to discriminate in the spectral domain. In this section, we see the role of the graph filter to make them local in the vertex or in the spectral domain.

- *Vertex Locality*: A graph filter is local in the vertex domain when the final filter output at each node is obtained by combining values from a local neighborhood around that node. For a Kth order filter, each node is influenced by its K-hop neighborhood. Thus, vertex locality needs a small filter order K.

- *Spectral Locality*: A graph filter is local in the frequency domain when its frequency response $h(\lambda)$ can isolate the components only around a specific frequency. It is similar to a narrow band-pass filter. To achieve this narrow-band filter, the filter order K needs to be high. Figure 4.5 shows the response of a filter local in the spectral domain.

- *Relation with filter order*: The above two definitions show that graph filters can be local in the vertex domain if the order is low but local in the spectral domain if the order is high. For example, let us consider the vertex-local graph filter $\mathbf{H}_1(\mathbf{S})$ with $K = 1$. In the frequency domain, the universal frequency response $h_1(\lambda)$ will be linear in λ. A linear function of λ cannot separate frequencies close to each other, as is shown in Figure 4.5. Thus $h_1(\lambda)$ and $\mathbf{H}_1(\mathbf{S})$ is not local in the spectral domain but local in the vertex domain. Differently, if we consider a response $h_2(\lambda)$, a polynomial in λ of order $K = 10$ which can separate λ_1 and λ_2, the corresponding vertex operator $\mathbf{H}_2(\mathbf{S})$ is not local in the vertex domain.

4.3. Applications

In this section, we look at two applications that utilize graph filter design in the frequency domain for two tasks, distributed signal denoising [1] and compressive spectral clustering [12].

4.3.1. Distributed Signal Denoising

In this section, we look at filter design for de-noising graph signals. The graph under consideration comprises 32 weather stations in the Molene region of France as nodes. Each station measures the temperature once every hour for 744 hours. The graph is built as follows: first, the similarity A_{ij} between each pair of stations is obtained as $A_{ij} = \exp(\frac{-d_{ij}}{\bar{d}})$ where d_{ij} is the physical distance between stations i and j and $\bar{d}$ is the average of all pair-wise distances; second, we obtain the 10 nearest neighborhood version of $\mathbf{A}$ as the desired graph. The graph signal $\mathbf{x} \in \mathbb{R}^{32}$ contains the temperature observed at all stations at a fixed hour. Signal $\mathbf{x}$ has similar

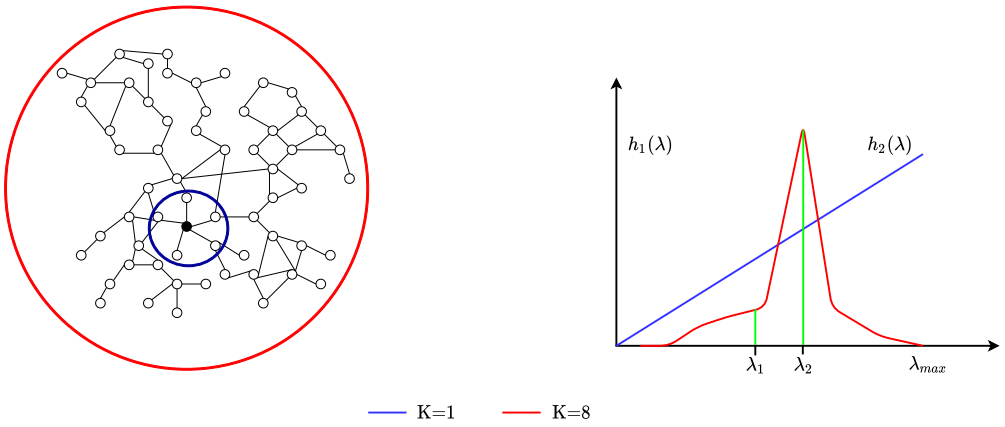


Figure 4.5: Non-local and local frequency response in the spectral domain. Frequencies λ_1, λ_2 are two closely spaced frequencies that need to be resolved. The linear response $h_1(\lambda)$ (blue) fails to resolve them, therefore being non-local in the frequency domain. The response $h_2(\lambda)$ corresponding to an order 8 filter (blue) is able to resolve them, therefore being localized in the frequency domain. However, in the vertex domain, their locality is showcased for the coloured node. $H_2(\mathbf{S})$ is not local in the vertex domain, with all nodes influencing the output at the selected node, whereas $H_1(\mathbf{S})$ is very localized.

values in adjacent nodes and thus we assume it is low-pass. Due to measurement noise, a noisy graph signal $\mathbf{y}$ is obtained as

$$\mathbf{y} = \mathbf{x} + \mathbf{n} \tag{4.25}$$

where $\mathbf{n}$ is i.i.d. zero-mean Gaussian with variance σ^2. Graph signal de-noising reconstructs the true signal $\mathbf{x}$ given the noisy observation $\mathbf{y}$. We want to solve this problem distributively. Graph convolutional filters are distributed operators in the vertex domain and we shall utilize them as the de-noising tool. The noise $\mathbf{n}$ is a high-pass graph signal; hence denoising implies low-pass graph filtering. The reconstructed graph signal $\mathbf{x}_r$ is obtained via filtering the output in the vertex domain as

$$\mathbf{x}_r = \sum_{k=1}^{K} h_k \mathbf{S}^k \mathbf{y} \tag{4.26}$$

where $\mathbf{S}$, $\mathbf{h} = [h_0, \ldots, h_K]^\top$ are the shift operator and filter coefficients, respectively. Let $\hat{\mathbf{y}}$ be the GFT of $\mathbf{y}$ and $\hat{\mathbf{x}}_r$ the GFT of the reconstructed $\mathbf{x}_r$. Since the desired output $\mathbf{x}$ is a low-pass graph signal, its GFT $\hat{\mathbf{x}}_r$ will be non-zero up to a critical frequency λ_c and zero beyond λ_c. We can write $\hat{\mathbf{x}}_r$ as

$$\begin{matrix} \hat{x}_r \neq 0, & \lambda_i \leq \lambda_l \\ \hat{x}_r = 0, & \lambda_i > \lambda_l \end{matrix}, \quad i = 1, \ldots, N \tag{4.27}$$

where $\lambda_1, \ldots, \lambda_N$ is the spectrum of $\mathcal{G}$. Since $\mathbf{x}_r$ is low-pass, our goal is to design a filter that is ideal up to λ_l and zero outside it. Let the desired frequency response be $\mathbf{g} = [g_1, \ldots, g_N]^\top$ with $g_i = 1$ if $\lambda_i \leq \lambda_l$ and zero otherwise. Using the relation between the input and output spectrum in (4.17), we can write for the ith eigenvalue the equation

$$g_i = \sum_{k=0}^{K} h_k \lambda_i^k \tag{4.28}$$

For $i = 1, \ldots, N$, the N equations can be written in the matrix format

$$\mathbf{g} = \mathbf{\Omega h} \tag{4.29}$$

where $\mathbf{\Omega}$ is the $N \times (K+1)$ Vandermonde-like matrix

$$\mathbf{\Omega} = \begin{pmatrix} 1 & \lambda_1 & \lambda_1^2 & \cdots & \cdots & \lambda_1^K \\ 1 & \lambda_2 & \lambda_2^2 & \cdots & \cdots & \lambda_2^K \\ 1 & \lambda_3 & \cdots & \cdots & \cdots & \vdots \\ \vdots & \cdots & \cdots & \cdots & \lambda_{N-1}^{K-1} & \lambda_{N-1}^K \\ 1 & \cdots & \cdots & \lambda_N^{K-2} & \lambda_N^{K-1} & \lambda_N^K \end{pmatrix} \tag{4.30}$$

Algorithm 2 Compressive Spectral Clustering

1. Set the number of clusters C and parameters $n = 2C\log C$, $d = 4\log(n)$, and $\gamma = 10^{-3}$;
2. Estimate eigenvalue λ_k of the Laplacian $\mathbf{L}$ via eigencount;
3. Generate a graph filter $\mathbf{H}_k(\mathbf{L})$ that approximates an ideal low-pass with cutoff frequency $_c = _k$;
4. generate d zero-mean random Gaussian graph signals with variance d^{-1} : $\mathbf{R} = [\mathbf{r}_1, \ldots, \mathbf{r}_d]^\top \in \mathbb{R}^{N\times d}$;
5. Filter all signals in $\mathbf{R}$ with $\mathbf{H}_k(\mathbf{L})$ and define the feature vector $\mathbf{f}_i \in \mathbb{R}^d$ for node i as

$$\tilde{f}_i = [(\mathbf{H}_k(\mathbf{L})\mathbf{R})^\top \boldsymbol{\delta}_i]/||(\mathbf{H}_k(\mathbf{L})\mathbf{R})^\top \boldsymbol{\delta}_i||_2$$

with δ_i being a Dirac vector;
6. Generate a random binary sampling matrix $\mathbf{M} \in \mathbb{R}^{n\times N}$ and keep only n feature vectors

$$[\tilde{\mathbf{f}}_1, \ldots, \tilde{\mathbf{f}_n}]^\top = \mathbf{M}[\mathbf{f}_1, \ldots, \mathbf{f}_N]^\top$$

7. Run k-means on the reduced features with Euclidean distance

$$D^r_{ij} = ||\tilde{\mathbf{f}}_i - \tilde{\mathbf{f}}_j||_w$$

to obtain k (one per cluster) reduced indicator vectors $\mathbf{c}^r_j \in \mathbb{R}^n$;
8. Interpolate each reduced indicator vector $\mathbf{c}^r_j$ by solving the optimization problem

$$\underset{\mathbf{x}\in\mathbb{R}^N}{\operatorname{argmin}} ||\mathbf{M}\mathbf{x} - \mathbf{c}^r_j||^2_2 + \gamma \mathbf{x}^\top \mathbf{G}(\mathbf{L})x$$

to obtain the k indicator vectors $\tilde{\mathbf{c}}^\star_j \in \mathbb{R}^N$ for all nodes.

and $\mathbf{h} = [h_0, \ldots, h_K]^\top$ is the vector of filter coefficients. We obtain $\mathbf{h}$ by solving the system of equations in (4.29). Since N is usually much larger than K, it is an over-determined system and the solution requires the pseudo-inverse of $\boldsymbol{\Omega}$ as

$$\mathbf{h} = \boldsymbol{\Omega}^\dagger \mathbf{g}. \tag{4.31}$$

This provides the filter coefficients for the filter to solve the de-noising task.

4.3.2. Compressive Spectral Clustering

Spectral clustering is a method to cluster the nodes of a graph, i.e., assign each node to a group. However, the preliminary approach in [5] requires the eigendecomposition of the graph Laplacian, which is a slow process for graphs having large number of nodes N. Compressive spectral clustering (CSC) [12] perform efficient spectral clustering without full eigendecomposition. The CSC algorithm is shown in 2. We focus on steps 3 and 5 of the algorithm. In spectral clustering, we are interested in the first few eigenvectors of the Laplacian, which are low-pass signals as mentioned before. To incorporate low-pass signals we need an ideal low-pass filter. In step 3, the ideal low-pass filter with cutoff frequency λ_k is approximated. The authors in [12] use Jackson-Chebyshev polynomials to perform this approximation [2]. This approximation takes the form

$$\mathbf{H}(\mathbf{L}) = \sum_{j=0}^{P} g^p_j \gamma_j T_j(\mathbf{L}) \tag{4.32}$$

where γ_j is the expansion coefficients of a step function and g_j is the jackson coefficient for the jth Chebyshev polynomial $T_j(\mathbf{L})$ that acts as a damping multiplier to counter Gibb's oscilaltions.
After finding the approximate response, the CSC algorithm filters random Gaussian signals in step 5, selects features at random to reduce the dimension in step 6, performs k-means clustering over them in step 7. Upon clustering, a membership signal for each cluster is interpolated over the graph by solving an optimization problem as shown in step 8. Since the approximation of the low-pass filter influences the rest of the operation, it is an important part of this algorithm.

4.4. Summary and Further Reading

More details on graph fourier transform can be found in [11, 7]. Frequency domain-based design of graph filters are present in [3, 4], and references therein. For adjacency matrix inspired frequency domain, refer to

the work in [7]. To learn more about Chebyshev approximation of finite impulse response and rational filters, please refer to [10] and [6] respectively.

Bibliography

[1] Siheng Chen et al. “Signal denoising on graphs via graph filtering”. In: *2014 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE. 2014, pp. 872–876.

[2] Edoardo Di Napoli, Eric Polizzi, and Yousef Saad. “Efficient estimation of eigenvalue counts in an interval”. In: *Numerical Linear Algebra with Applications* 23.4 (2016), pp. 674–692.

[3] Elvin Isufi et al. “Autoregressive moving average graph filtering”. In: *IEEE Transactions on Signal Processing* 65.2 (2016), pp. 274–288.

[4] Jiani Liu, Elvin Isufi, and Geert Leus. “Filter design for autoregressive moving average graph filters”. In: *IEEE Transactions on Signal and Information Processing over Networks* 5.1 (2018), pp. 47–60.

[5] Andrew Y Ng, Michael I Jordan, Yair Weiss, et al. “On spectral clustering: Analysis and an algorithm”. In: *Advances in neural information processing systems* 2 (2002), pp. 849–856.

[6] Oxana Rimleanscaia and Elvin Isufi. “Rational Chebyshev Graph Filters”. In: *Asilomar 2020*. IEEE. 2011, pp. 1–8.

[7] Aliaksei Sandryhaila and Jose MF Moura. “Discrete signal processing on graphs: Frequency analysis”. In: *IEEE Transactions on Signal Processing* 62.12 (2014), pp. 3042–3054.

[8] Stefania Sardellitti, Sergio Barbarossa, and Paolo Di Lorenzo. “On the graph Fourier transform for directed graphs”. In: *IEEE Journal of Selected Topics in Signal Processing* 11.6 (2017), pp. 796–811.

[9] Rasoul Shafipour et al. “A directed graph Fourier transform with spread frequency components”. In: *IEEE Transactions on Signal Processing* 67.4 (2018), pp. 946–960.

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

[11] David I Shuman et al. “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains”. In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[12] Nicolas Tremblay et al. “Compressive spectral clustering”. In: *International conference on machine learning*. PMLR. 2016, pp. 1002–1011.

5

Edge Varying Graph Filters

Previous chapters focused on FIR graph filters. They have K scalar filter coefficients for an order-K FIR filter, while their expressive power is limited as well due to this small number of parameters. In this chapter, we focus on more expressive graph filters known as edge varying graph filters.

5.1. Edge varying graph filters

Global parameters. A popular graph filter we defined in Chapter 8 is the graph convolutional filters, which is a polynomial of the graph shift operator $\mathbf{S} \in \mathbb{R}^{N \times N}$,

$$\mathbf{H} \triangleq \sum_{k=0}^{K} h_k \mathbf{S}^k. \tag{5.1}$$

When this type of filters is operated on a graph signal $\mathbf{x} \in \mathbb{R}^N$, the filter output $\mathbf{y} \in \mathbb{R}^N$ has the form

$$\mathbf{y} = \mathbf{H}\mathbf{x} = \sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x} = \sum_{k=0}^{K} h_k \mathbf{S}\mathbf{x}^{(k-1)} = \sum_{k=0}^{K} h_k \mathbf{x}^{(k)}, \tag{5.2}$$

where $\mathbf{x}^{(k)} = \mathbf{S}^k\mathbf{x} = \mathbf{S}\mathbf{x}^{(k-1)}$ is the k-th shifted version of the graph signal. An order $K = 4$ graph convolutional filter output is illustrated in Figure 5.1. Specifically, on node i, the output signal value, y_i, can be expressed as

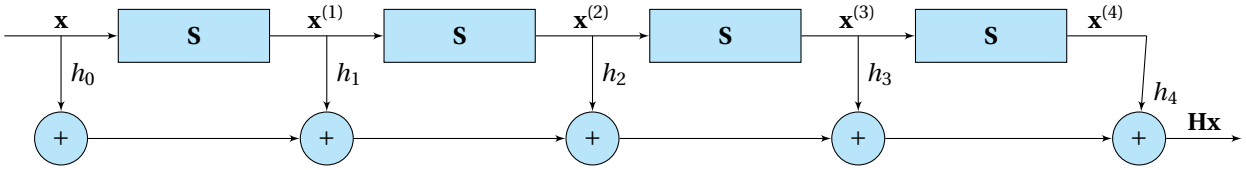


Figure 5.1: Schematic illustration of a graph convolutional filter output of order $K = 4$.

a local combination of the neighbors, i.e.,

$$y_i = \sum_{k=0}^{K} h_k \sum_{j \in \mathcal{N}_i} [\mathbf{S}^k]_{i,j} x_j = \sum_{k=0}^{K} h_k \sum_{j \in \mathcal{N}_i} S_{i,j} x_j^{(k-1)}, \tag{5.3}$$

where $x_j^{(k-1)}$ is the signal value on node j of the $(k-1)$-th shifted signal $\mathbf{x}^{(k-1)}$, $\mathcal{N}_i$ indicates the neighbourhood of node i, and $[\mathbf{S}^k]_{i,j}$ is the entry (i,j) of the k-th order graph shift operator. From (5.3), we see that the coefficients h_k of FIR filters are shared among all nodes. That is, filter coefficient h_k is the same for all nodes $i \in \mathcal{V}$. For each k-th shifting operation, the signals coming from the neighbourhood are aggregated based on the edge weights of the graph, then weighted commonly by filter coefficient h_k. We call h_k a global parameter.

The global parameters used in (5.1) are useful to transfer the filter among different graphs without keeping track of node labeling, However, they limit the expressive power of these filters. In the following, we will introduce another family of graph filters that uses instead local parameters.

5.1.1. Constrained edge varying graph filters

Let us first consider a constrained edge varying graph filter, where at each k-th order shifting $\mathbf{S}^k$, we replace the scalar filter coefficient h_k in (5.1) by an edge varying filter coefficient $\mathbf{\Phi}^{(k)} \in \mathbb{R}^{N\times N}$. As a result, a constrained edge varying graph filter has the form

$$\mathbf{H}_{\mathrm{c}} = \mathbf{\Phi}^{(1)} + \mathbf{\Phi}^{(2)}\mathbf{S} + \cdots + \mathbf{\Phi}^{(K)}\mathbf{S}^{K-1} \triangleq \sum_{k=1}^{K} \mathbf{\Phi}^{(k)}\mathbf{S}^{k-1}, \tag{5.4}$$

where matrices $\mathbf{\Phi}^{(k)}$ share the support with $\mathbf{S}+\mathbf{I}_N$ and are called edge varying filter coefficient matrices. For an input graph signal $\mathbf{x}$, the filter output is

$$\mathbf{y}_{\mathrm{c}} = \mathbf{H}_{\mathrm{c}}\mathbf{x} = \sum_{k=1}^{K} \mathbf{\Phi}^{(k)}\mathbf{S}^{k-1}\mathbf{x} = \sum_{k=1}^{K} \mathbf{\Phi}^{(k)}\mathbf{x}^{(k-1)}. \tag{5.5}$$

This operation is illustrated in Figure 5.2. From a node perspective, the filter output y_i on node i has the form

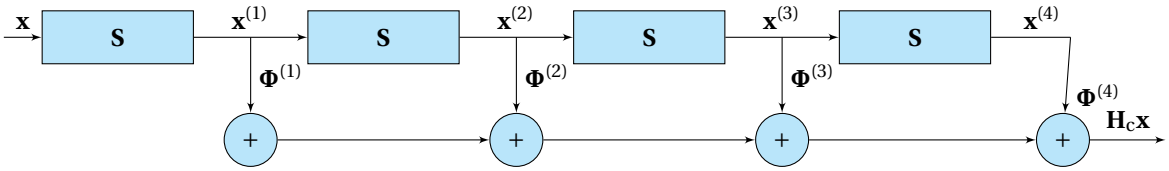


Figure 5.2: Schematic illustration of the constrained edge varying graph filter output of order $K = 4$ [2].

$$y_{\mathrm{c},i} = \sum_{k=1}^{K} \sum_{j\in\mathcal{N}_i} [\mathbf{\Phi}^{(k)}\mathbf{S}]_{i,j} x_j^{(k-2)} = \sum_{k=1}^{K} \sum_{j\in\mathcal{N}_i} \Phi_{i,j}^{(k)} x_j^{(k-1)}. \tag{5.6}$$

Equivalently, each node can compute locally the filter output by tracking the following quantities:

- the regular shift output $\mathbf{x}^{(k)} = \mathbf{S}\mathbf{x}^{(k-1)}, \mathbf{x}^{(0)} = \mathbf{x}$;
- the edge weighted shift output $\mathbf{z}^{(k)} = \mathbf{\Phi}^{(k)}\mathbf{x}^{(k-1)}$;
- the accumulator output $\mathbf{y}_{\mathrm{c}}^{(k)} = \mathbf{y}_{\mathrm{c}}^{(k-1)} + \mathbf{z}^{(k)}, \mathbf{y}^{(0)} = \mathbf{0}$.

Note that since the edge varying weight matrix $\mathbf{\Phi}^{(k)}$ enjoys the same sparsity as the shift operator $\mathbf{S}$, both $\mathbf{x}^{(k)}$ and $\mathbf{z}^{(k)}$ require only neighboring information. The final filter output is $\mathbf{y} = \mathbf{y}^{(K)}$ which yields a computational complexity same as classical FIR filter, $\mathcal{O}(MK)$, where M is the number of edges in the graph [1].

If we compare the constrained edge varying graph filter with the conventional FIR filter from a node perspective as in (5.3) and (5.6), after signal shifting $\mathbf{S}$k, the conventional FIR filter weights each shifted graph signal by a common scalar h_k, while the constrained edge varying filter mixes $\mathbf{x}^{(k-1)}$ locally using an edge dependent weight matrix $\mathbf{\Phi}^{(k)}$. We call this a local parameter. Note that the conventional FIR graph filter can be considered as a special case of the constrained edge varying filter, when $\mathbf{\Phi}^{(k)} = h_k\mathbf{S}$ for each k.

5.1.2. Edge varying graph filters

The more general edge varying graph filter can be defined as follows

$$\mathbf{H}_{\mathrm{ev}} = \mathbf{\Phi}^{(1)} + \mathbf{\Phi}^{(2)}\mathbf{\Phi}^{(1)} + \cdots + \mathbf{\Phi}^{(K)}\mathbf{\Phi}^{(K-1)}\ldots\mathbf{\Phi}^{(1)} \triangleq \sum_{k=1}^{K} \mathbf{\Phi}^{(k:1)}, \tag{5.7}$$

with product matrices $\mathbf{\Phi}^{(k:1)} = \prod_{k'=1}^{k} \mathbf{\Phi}^{(k')} = \mathbf{\Phi}^{(K)}\mathbf{\Phi}^{(K-1)}\ldots\mathbf{\Phi}^{(1)}$ and $\mathbf{\Phi}^{(k')} \in \mathbb{R}^{N\times N}$, for all $k' = 1,\ldots,K$ is an edge varying weighting matrix with the same support as $\mathbf{S}+\mathbf{I}_N$. If a graph signal $\mathbf{x}$ is fed in this filter, the output is

$$\mathbf{y}_{\mathrm{ev}} = \sum_{k=1}^{K} \mathbf{z}^{(k)} = \sum_{k=1}^{K} \mathbf{\Phi}^{(k:1)}\mathbf{x}, \tag{5.8}$$

with the intermediate output

$$\mathbf{z}^{(k)} \triangleq \mathbf{\Phi}^{(k:1)}\mathbf{x} = \prod_{k'=1}^{k} \mathbf{\Phi}^{(k')}\mathbf{x} \tag{5.9}$$

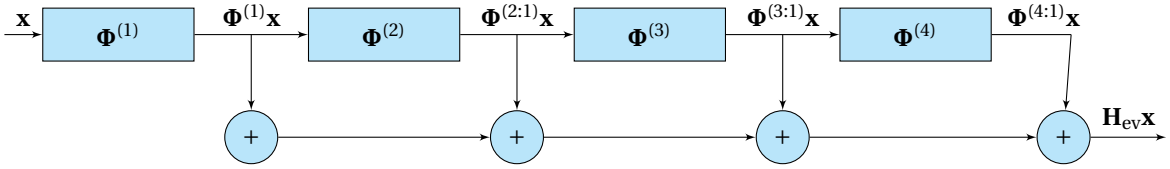


Figure 5.3: Schematic illustration of the edge varying graph filter output of order $K = 4$ [2].

for $k = 1, \dots, K$. The operation (5.8) is illustrated in the Figure 5.3. Note that signal $\mathbf{z}^{(k)}$ can be computed using the recursion

$$\mathbf{z}^{(k)} = \mathbf{\Phi}^{(k)}\mathbf{z}^{(k-1)}, \text{ for } k = 1, \dots, K \tag{5.10}$$

with initialization $\mathbf{z}^{(0)} = \mathbf{x}$. This recursive expression implies signal $\mathbf{z}^{(k)}$ is produced from $\mathbf{z}^{(k-1)}$ using operations that are local in the graph. Indeed, since $\mathbf{\Phi}^{(k)}$ shares the sparsity pattern of $\mathbf{S} + \mathbf{I}_N$, node i computes its component $z_i^{(k)}$ as

$$z_i^{(k)} = \sum_{j \in \mathcal{N}_i} \Phi_{i,j}^{(k)} z_j^{(k-1)}. \tag{5.11}$$

When $k = 1$, it follows that each node i builds the i-th entry as the values of the signal $\mathbf{x}$ at most at neighbouring nodes. Particularizing $k = 2$, (5.11) shows the components of $\mathbf{z}^{(2)}$ depend only on the values of signal $\mathbf{z}^{(1)}$ at neighboring nodes which, in turn, depend only on the values of $\mathbf{x}$ at their neighbors. Thus, the components of $\mathbf{z}^{(2)}$ are a function of the values of $\mathbf{x}$ at most at the respective two-hop neighbours. Repeating this argument iteratively, $z_i^{(k)}$ represents an aggregation of information at node i coming from its k-hop neighborhood.

The collection of signals $\mathbf{z}^{(k)}$ in (5.10) behaves like a sequence of scaled shift operations through the graph (the signal values are shifted between neighboring nodes). The filter output in (5.8) can be understood by interpreting $\mathbf{\Phi}^{(k:1)}$ as a scaled shift, which holds because of its locality. Each shift $\mathbf{\Phi}^{(k:1)}$ is a recursive composition of individual shifts $\mathbf{\Phi}^{(k)}$. These individual shifts represent different operators that respect the structure of $\mathcal{G}$ while reweighing individual edges differently when needed. We call matrix $\mathbf{\Phi}^{(k)}$ local shift parameters.

Edge varying graph filter is a distributed graph filter. To compute the output in (5.8), each node is only required to track the following quantities:

- the shifted signal output $\mathbf{x}^{(k)} = \mathbf{\Phi}^{(k)}\mathbf{x}^{(k-1)}, \mathbf{x}^{(0)} = \mathbf{x}$,
- the accumulator output $\mathbf{y}^{(k)} = \mathbf{y}^{(k-1)} + \mathbf{x}^{(k)}, \mathbf{y}^{(0)} = \mathbf{0}$.

Both these operations can be computed locally in each node by combining only neighboring data. Hence, edge varying filtering operation (5.8) preserves the efficient distributed implementation of the classical graph filter (5.1) with a complexity of $\mathcal{O}(MK)$, where M is the number of edges in the graph. We comment that constrained edge varying graph filter is a special case of edge varying graph filter.

5.1.3. Remarks

We group here a few remarks regarding the filters discussed above.

- Both constrained edge varying and edge varying graph filters in (5.5) and (5.8) have a computational complexity of $\mathcal{O}(MK)$, which is the same as the classical FIR graph filter. This comes from the fact that matrices $\mathbf{\Phi}^{(k)}$ share the support with $\mathbf{S} + \mathbf{I}_N$.
- The number of parameters in both the constrained edge varying and the edge varying graph filter is the same as the complexity order, $\mathcal{O}(K(M+N))$, since each edge varying weight matrix $\mathbf{\Phi}^{(k)}$ shares the support of $\mathbf{I}_N + \mathbf{S}$.

  In the classical FIR filter, the number of parameters is K, and (5.1) forces all nodes to weigh the information of all k-hop neighbors with the same parameter h_k. The edge varying type filters we introduced here have more parameters by introducing a matrix weight instead of a scalar weight. This matrix weight has the ability to assign different weights to the information coming from different neighbors. Specifically, each node i in (5.11) has a different parameter $\Phi_{i,j}^{(k)}$ for each neighbor j.

- Consider we are interested in implementing distributively a linear operator $\tilde{\mathbf{H}}$ with graph filters. If we use a FIR filter as in (5.1), the filter order K needs to be large if a high accuracy is required. As the computational complexity scales with K, large-order graph filters incur high costs. The edge varying graph filter, due to its enhanced degrees of freedom can approximate $\tilde{\mathbf{H}}$ with a lower order K. Therefore, it leads to a more efficient implementation.
- The constrained edge varying filter has often better numerical implementation properties than the full edge varying filter [1]. In the constrained edge varying filter (5.5), instead of adopting a different diffusion matrix at every step as in full edge varying form (5.8), the signal diffusion occurs through the GSO $\mathbf{S}$.
- Full edge varying graph filter is more useful when the topological weights are unknown. This is because it only needs the support and allocates the weights to fit the task at hand.
- From the perspective of independence of node labeling, the FIR graph filter in shown to be permutation equivariant, so it can be applied to any graph without knowing the specific node labeling. However, the edge varying graph filter cannot as edge weights are allocated to specific couples. This is the price to pay for increasing the degrees of freedom.

5.2. Hybrid graph filter

We can consider what we call hybrid filters defined as a combination of classical and edge varying graph filters. Formally, let $\mathcal{I} \subset \mathcal{V}$ denote an importance subset of $I = |\mathcal{I}|$ nodes and define shift matrices $\mathbf{\Phi}_{\mathcal{I}}^{(k)}$ such that $[\mathbf{\Phi}_{\mathcal{I}}^{(k)}]_{i,j} = 0$ for all $i \notin \mathcal{I}$ or $(i,j) \notin \mathcal{E}$ and $k \geq 1$. That is, the edge varying coefficient matrices $\mathbf{\Phi}_{\mathcal{I}}^{(k)}$ may contain nonzero elements only at rows i that belong to set $\mathcal{I}$ and with the node j being a neighbor of i. We define hybrid filters as those of the form

$$\mathbf{H}_{\text{h}} = \sum_{k=1}^{K} \Big(\prod_{k'=1}^{k} \mathbf{\Phi}_{\mathcal{I}}^{(k')} + h_k \mathbf{S}^k \Big). \tag{5.12}$$

In essence, nodes $i \in \mathcal{I}$ learn edge dependent parameters which may also be different at different nodes, while the remaining nodes $i \notin \mathcal{I}$ learn global parameters h_k as in (5.1).

Hybrid filters are defined by a number of parameters that depend on the total number of neighbours of all nodes in the important set $\mathcal{I}$. Define the sum of the numbers of their neighbors $M_{\mathcal{I}} = \sum_{i \in \mathcal{I}} |\mathcal{N}_i|$ and observe that $\mathbf{\Phi}_{\mathcal{I}}^{(k)}$ for $k \geq 1$ have respectively $M_{\mathcal{I}}$ nonzero values. We then have $KM_{\mathcal{I}}$ parameters in the edge varying filter and K parameters in the classical filter. The implementation cost of a hybrid graph filter is of order $\mathcal{O}(KM)$ since both terms in (5.12) respect the sparsity of the graph [1].

Consider an example in Figure.5.4. The nodes in set $\mathcal{I} = \{2,7\}$ are highlighted. Nodes 2 and 7 have edge varying parameters associated to their incident edges. All nodes, including 2 and 7, also use the global parameter h_k as in a regular graph filter (5.1) [2].

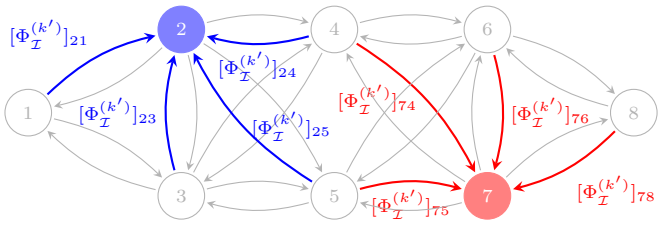


Figure 5.4: Hybrid edge varying filter [cf. (5.12)].

5.3. Node varying graph filter

When the degrees of freedom of the above filters result too many for a given dataset, we can further restrict them but still be more general than the classical filter. This can be achieved by allocating node-specific coefficients [3]. In the node domain, these filters have the form

$$\mathbf{H}_{\text{nv}} \triangleq \sum_{k=0}^{K} \text{diag}(\boldsymbol{\phi}_k) \mathbf{S}^k, \tag{5.13}$$

where vector $\boldsymbol{\phi}_k = [\phi_{k,1}, \ldots, \phi_{k,N}]^\top$ contains the node dependent coefficients applied at the k-th shift. That is, at each node, it applies a different coefficient. When $\mathbf{H}_{\text{nv}}$ is applied to a signal $\mathbf{x}$, each node applies different weights to the shifted signals $\mathbf{x}^{(k)}$. The output $\mathbf{y}$ can be expressed as

$$\mathbf{y}_{\text{nv}} = \sum_{k=0}^{K} \text{diag}(\boldsymbol{\phi}_k)\mathbf{S}^k\mathbf{x} = \sum_{k=0}^{K} \text{diag}(\boldsymbol{\phi}_k)\mathbf{x}^{(k)}. \tag{5.14}$$

This operation is illustrated in Figure 5.5. At node i, the output signal value is

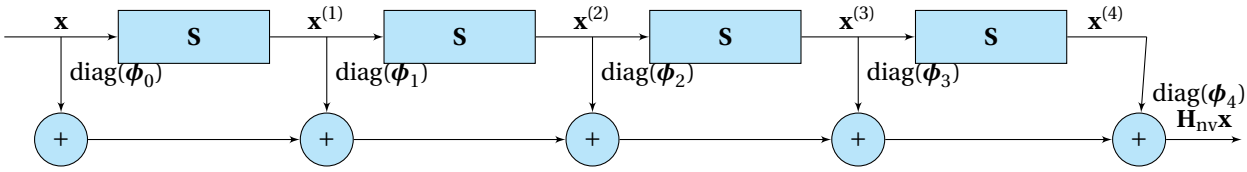


Figure 5.5: Schematic illustration of a node varying graph filter output of order $K = 4$.

$$y_{\text{nv},i} = \sum_{k=0}^{K} \sum_{j \in \mathcal{N}_i} [\mathbf{S}^k]_{i,j} \phi_{k,j} x_j = \sum_{k=0}^{K} \sum_{j \in \mathcal{N}_i} [\mathbf{S}]_{i,j} \phi_{k,j} x_j^{(k)}, \tag{5.15}$$

where we can see that in node varying graph filtering, each node signal is weighted differently. This additional flexibility enables the design of more general operators without undermining the local implementation.

The classical FIR filter can be considered as a special case when $\boldsymbol{\phi}_k = h_k \mathbf{1}$ in node varying graph filters. The node varying FIR filter preserves also the efficient implementation as classical FIR filter since it applies the node coefficients to the kth shifted input $\mathbf{S}^k\mathbf{x} = \mathbf{S}(\mathbf{S}^{k-1}\mathbf{x})$ with a computational complexity of $\mathcal{O}(KM)$. In turn, node varying graph filters can be seen as a special case of edge varying graph filters when $\boldsymbol{\Phi}^{(k)} = \text{diag}(\boldsymbol{\phi}_k)$ in constrained edge varying filter.

5.4. Summary and Further Reading

In this chapter, we have introduced several variants or generalizations of the FIR graph filters, which have more expressive power compared to the FIR graph filters. Instead of relying on a common scalar weight as the filter coefficient, these graph filters are built based on varying the filter coefficients in a node-wise or edge-wise fashion. In addition, we also study the distributed implementation and the computational complexity for each corresponding filtering operation. We refer readers to [1] for more details in generalized graph filters.

Bibliography

[1] Mario Coutino, Elvin Isufi, and Geert Leus. “Advances in distributed graph filtering”. In: *IEEE Transactions on Signal Processing* 67.9 (2019), pp. 2320–2333.

[2] Elvin Isufi, Fernando Gama, and Alejandro Ribeiro. “Edgenets: Edge varying graph neural networks”. In: *arXiv preprint arXiv:2001.07620* (2020).

[3] Santiago Segarra, Antonio G Marques, and Alejandro Ribeiro. “Optimal graph-filter design and applications to distributed linear network operators”. In: *IEEE Transactions on Signal Processing* 65.15 (2017), pp. 4117–4131.

6

Empirical Risk Minimization

In this chapter, we introduce statistical learning problem and move the Empirical Risk Minimization (ERM) problem. Afterward, we discuss different machine learning paradigms, and introduce neural network models from an ERM perspective.

6.1. Basic Elements

6.1.1. Statistical Learning

Statistical learning relates an input $\mathbf{x}$ and output $\mathbf{y}$ through a joint probability distribution $p(\mathbf{x},\mathbf{y})$. Hence, the task is to predict $\mathbf{y}$, given a known input $\mathbf{x}$, with a conditional distribution $\mathbf{y} \sim p(\mathbf{y}\,|\,\mathbf{x})$. However, in the most applications a deterministic output is desired and the output is estimated via the conditional expectation $\mathbf{y} = \mathbb{E}\left[\mathbf{y}\,|\,\mathbf{x}\right]$. The goal of a learning problem is to identify the function $\phi(\mathbf{x})$ that best estimates the output variable $\mathbf{y}$ given an input variable $\mathbf{x}$ drawn from the distribution $p(\mathbf{x},\mathbf{y})$. This can be achieved by minimizing the estimation error in a statistical sense as we explain next. Fig. 6.1 illustrates the link between input $\mathbf{x}$, desired output $\mathbf{y}$, and estimated output $\hat{\mathbf{y}}$.

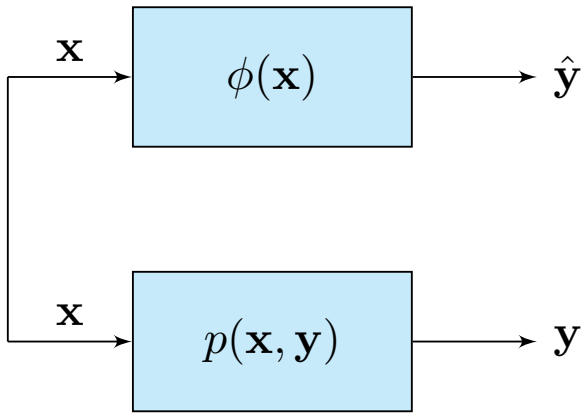


Figure 6.1: The link between input $\mathbf{x}$, desired output $\mathbf{y}$, and estimated output $\hat{\mathbf{y}}$.

6.1.2. Statistical Risk Minimization

There are three main elements in a learning problem: a probabilistic model (distribution) $p(\mathbf{x},\mathbf{y})$ that generates the data, a family (class) of functions $\mathscr{C}$ that defines the space of functions $\phi(\cdot)$, and a loss function $l(\phi(\mathbf{x}),\mathbf{y})$ that measures how well function $\phi(\mathbf{x})$ predicts $\mathbf{y}$. Considering these elements, the Statistical Risk Minimization (SRM) learns the optimal estimator by minimizing the expected loss over the distribution as

$$\phi^{\star} = \underset{\phi\in\mathscr{C}}{\operatorname{argmin}}\, \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[l(\mathbf{y},\phi(\mathbf{x}))\right]. \tag{6.1}$$

The function $\phi^\star(\cdot)$ is a deterministic mapping between input $\mathbf{x}$ and the most expected corresponding output based on the joint distribution $p(\mathbf{x},\mathbf{y})$. Typical examples for loss function are quadratic loss $l(\hat{\mathbf{y}},\mathbf{y}) = \|\hat{\mathbf{y}}-\mathbf{y}\|_2^2$ for regression problems, and indicator function $l(\hat{y},y) = \mathbf{I}(\hat{y}=y)$ for binary classification problems. Then, the goal of a machine learning scientist is to define the loss function $l(\cdot)$ and the class of function $\mathscr{C}$ based on the prior knowledge of the problem. Once the SRM problem solution from (6.1) is obtained, function $\phi^\star(\cdot)$ is used at test time for any input data $\mathbf{x}$ from the distribution. Fig. 6.2 illustrates the block diagram for the SRM problem.

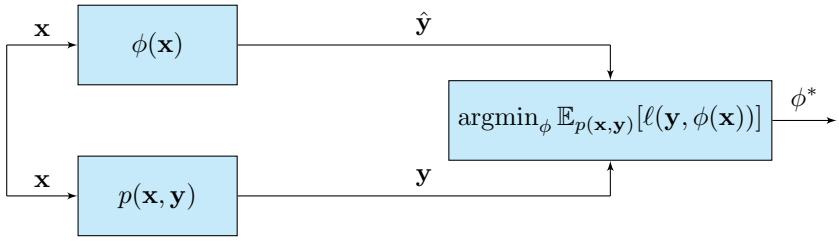


Figure 6.2: The block diagram of the SRM problem.

The main drawback of the SRM problem is that the probabilistic model of the data $p(\mathbf{x},\mathbf{y})$ is usually unknown and only a subset of data pairs is available. Hence, to tackle this problem, it is necessary to move from statistical risk into empirical risk problem [5].

6.1.3. Empirical Risk Minimization

In many of real-world problems, instead of data model $p(\mathbf{x},\mathbf{y})$, we observe a set of data pairs called *training set* $\mathscr{T} = \{(\mathbf{x}_i,\mathbf{y}_i)\}_{i=1}^m$. Then, the Empirical Risk Minimization (ERM) problem can be defined as approximating the SRM problem in (6.1) by replacing the mathematical expectation on $p(\mathbf{x},\mathbf{y})$ with sample average over the training set $\mathscr{T}$ as

$$\phi^\star = \underset{\phi\in\mathscr{C}}{\text{argmin}} \frac{1}{m}\sum_{i=1}^m l(\mathbf{y}_i,\phi(\mathbf{x}_i)). \tag{6.2}$$

It is obvious that the statistical risk has changed to empirical risk, and therefore, the optimal function $\phi^\star(\mathbf{x})$ can only learn from the training set data now. Fig (6.3) indicates the block diagram of the ERM problem. As you can see, the data comes from training set instead of model distribution, and hence, the mathematical expectation should be replaced by empirical average.

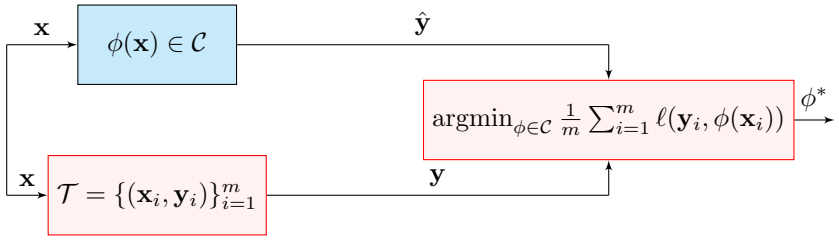


Figure 6.3: The block diagram of the ERM problem.

Although the ERM problem minimizes the empirical loss, it still aims to reduce the *generalization error* $\mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[l(\mathbf{y},\phi(\mathbf{x}))\right]$ which also indicates the performance of the optimal estimator $\phi^\star(\mathbf{x})$ over new data that are not in $\mathscr{T}$. In other words, minimizing the empirical risk alone will not lead to a good optimal function, since at some point the function will *overfit* the training data leading to a poor performance on the unseen data. In summary, we want that the optimal function $\phi^\star(\cdot)$ learns the training data well in learning time, and also generalizes this good performance on new data in test time.

6.2. Parametric ERM

To constraint the space of functions in the ERM problem we use *parametric* functions which means each function in the family can be represented by a set of parameters $\mathscr{W}$. Then the family $\mathscr{C}$ can be shown by a set

of parameters $\mathscr{W}$. For example we can consider linear parametric function set $\phi(\mathbf{x}) = \mathbf{W}\mathbf{x}$, where $\mathbf{W} \in \mathscr{W}$ is a matrix of parameters. The new problem is called now *Parametric ERM* and defined as

$$\mathscr{W}^\star = \underset{\mathscr{W}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{y}_i, \phi_{\mathscr{W}}(\mathbf{x}_i)), \tag{6.3}$$

where $\phi_{\mathscr{W}}(\cdot)$ is a function parameterized by $\mathscr{W}$. Now, the family of function should be selected so that the parametric ERM provides a good approximation of the SRM problem. Moreover, moving into parametric function classes allows us to use regularization terms for preventing overfitting or forcing a prior knowledge

$$\mathscr{W}^\star = \underset{\mathscr{W}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{y}_i, \phi_{\mathscr{W}}(\mathbf{x}_i)) + g(\mathscr{W}), \tag{6.4}$$

where $g(\mathscr{W})$ is the regularization term to improve generalization by handling a trade-off between empirical risk and model complexity. For example, the regularized parametric ERM problem in (6.4) for linear function class, quadratic loss, and *Tikhonov* regularization is

$$\mathbf{W}^\star = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} \|\mathbf{y}_i - \mathbf{W}\mathbf{x}_i\|_2^2 + \|\mathbf{W}^2\|_F, \tag{6.5}$$

The question of how good is this approximation has been studied in the context of statistical learning and interested readers can refer to [3, 5]. Moreover, the parametric ERM problem is usually minimized by *stochastic gradient descent* algorithm or its variants during training.

6.3. ERM and Machine Learning Problems

As we saw above, the ERM problem relies on three main aspects: the family of functions $\mathscr{C}$; the loss function $l(\cdot)$; and the training set $\mathscr{T}$. Depending on the nature of the training set, the ERM problem may be approaches from a supervised learning or an unsupervised learning perspective. In this section, we will concretize the latter.

6.3.1. Supervised learning

Supervised learning aims to learn a mapping between input-output pairs from a set of inputs and their corresponding desired outputs called training set $\mathscr{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^m$. Hence, generally, in the supervised learning problem, the goal is to find an optimal function $\phi_{\mathscr{W}}(\mathbf{x})$ which minimizes the average loss (empirical risk) over the training set $\mathscr{T}$, and it can be defined as a parametric ERM problem

$$\phi_{\mathscr{W}}^\star = \underset{\phi_{\mathscr{W}} \in \mathscr{C}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{y}_i, \phi_{\mathscr{W}}(\mathbf{x}_i)), \tag{6.6}$$

where the loss function $l(\cdot)$ and class of functions $\mathscr{C}$ can be selected by the goals and prior knowledge of the problem.

6.3.2. Unsupervised leaning

In unsupervised learning problem, the data set has no pre-existing label, $\mathscr{T} = \{\mathbf{x}_i\}_{i=1}^m$, and the goal is to extract hidden patterns and similarities in the data set. Again, although there are no labels, the unsupervised learning problem can be defined as an ERM problem

$$\phi_{\mathscr{W}}^\star = \underset{\phi_{\mathscr{W}} \in \mathscr{C}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\phi_{\mathscr{W}}(\mathbf{x}_i)), \tag{6.7}$$

where the loss function $l(\cdot)$ should be selected based on the specific task.

6.3.3. Least square problem as ERM for a linear model

One of the typical machine learning regression problems is the least square problem. It is interesting that this problem can also be shown as ERM for the space of linear functions. Consider the parametric ERM problem in (6.3) for scalar output with loss function: $l(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$, and assume a linear model for the family of functions: $\phi(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$. The ERM problem for this loss function and class of functions can be rewritten as

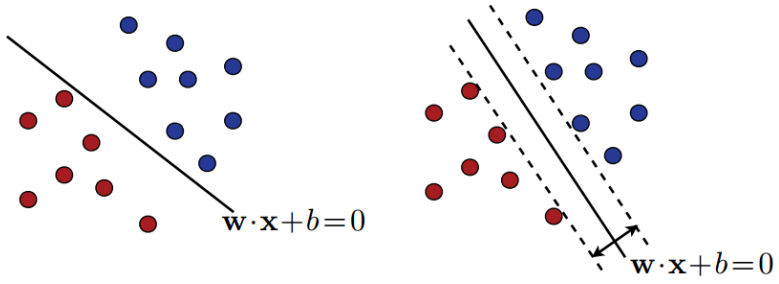


Figure 6.4: Two possible linear classification. The left-hand side is a simple linear separation, and the right-hand side maximizes the margin as the SVM does [3].

$$\mathbf{W}^\star = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m}(\mathbf{W}^T\mathbf{x}_i - y_i)^2, \tag{6.8}$$

where $\mathbf{W}\in\mathbb{R}^n$ is the vector of model parameters. Consider $\mathbf{x}=[\mathbf{x}_1,\ldots,\mathbf{x}_m]\in\mathbb{R}^{n\times m}$ as the matrix of all input vectors, and $\mathbf{y}=\left[y_1,\ldots,y_m\right]^\top$ as the vector of all output labels in the data set. Now, By rewriting (6.8) in vector notation, we have

$$\mathbf{W}^\star = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{m}\|\mathbf{x}^T\mathbf{W}-\mathbf{y}\|_2^2. \tag{6.9}$$

Finally, it is obvious that the ERM problem for linear model and quadratic loss in (6.9) is equal to least square problem, where model parameters $\mathbf{W}$ correspond to unknown variables vector in the least square problem. It should be also noted that the constant multiplier $\frac{1}{m}$ cannot change the minimizer and will not affect the optimal solution.

6.3.4. Support Vector Machine (SVM)

In classification problems, two sets of n-dimensional points are given and we seek to find a function $f:\mathbb{R}^n\longrightarrow\mathbb{R}$ that is positive on the first set and negative on the second one. The linear classification problem evolves when we consider a family of linear functions $f(\mathbf{x})=\mathbf{W}^T\mathbf{x}+b$ as the discriminator. The SVM algorithm is a stable approach for solving linear classification problem since it looks for a linear separator that maximizes margin. More formally, consider the data set $\mathscr{T}=\{(\mathbf{x}_i,y_i)\}_{i=1}^m$ where $\mathbf{x}_i$ is a data point and $y_i\in\{-1,+1\}$ is its corresponding label. Furthermore, by assuming the linear separator $f(\mathbf{x})=\mathbf{W}^T\mathbf{x}+b$, the *geometric margin for each data point* can be defined as

$$\rho_h(\mathbf{x}) = \frac{\|\mathbf{W}^T\mathbf{x}+b\|}{\|\mathbf{W}\|}, \tag{6.10}$$

and then, the *geometric margin of classifier* defines as

$$\rho_h = \min_{i\in\{1,\ldots,m\}} \rho_h(\mathbf{x}_i), \tag{6.11}$$

that shows the distance between the linear function and closest data point. As shown in Fig. 6.4, unlike the simple linear separation, the SVM algorithm chooses a specific discriminator hyperplane with maximum margin that provides the algorithm robustness and better generalization.

With a normalization, the margin will be $\rho_h = 1/\|\mathbf{W}\|_2$. An illustration for this normalization is depicted in Fig. 6.5. The separable SVM tries to maximize this margin by keeping the empirical error equal to zero through the following optimization problem

$$\begin{aligned}(\mathbf{W}^\star,b^\star) = \underset{\mathbf{W},b}{\operatorname{argmin}} \quad & \frac{1}{2}\|\mathbf{W}\|_2^2,\\ \text{subject to} \quad & y_i(\mathbf{W}^T\mathbf{x}_i+b)\geq 1 \quad \text{for all} i\in\{1,\ldots,m\}.\end{aligned} \tag{6.12}$$

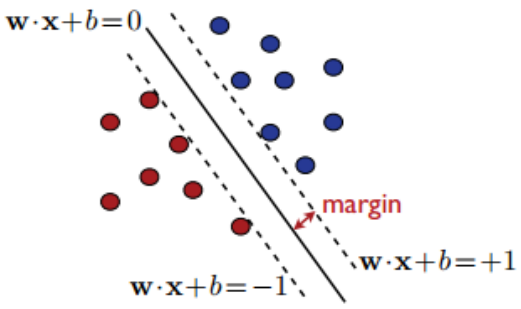


Figure 6.5: Maximum-margin hyperplane with normalization of model parameters. The marginal hyperplanes are shown with dashed lines [3].

This problem can be seen as a regularized parametric ERM problem on linear functions. The loss function appeared as a constraint and does not allow any miss-classification on training data. Also, the margin is used for regularization and provides robustness for the SVM.

6.4. ERM and Neural Networks

Neural networks are information processing architecture formed by cascaded layers. In this section, we will approach them from an ERM perspective. In neural networks, some hidden layers are added between the input and output layers to create a more complicated space of functions. It should be noted that it is pointless to concatenate linear models for the sake of hidden layers and a non-linearity or activation function is required between layers.

Consider the data set $\mathcal{T} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^m$ containing m input-output pairs of data, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{y} \in \mathbb{R}^p$, and loss function $l(\cdot)$. The building block of neural networks is the *perceptron* which is defined as

$$\mathbf{y} = \sigma(\mathbf{W}\mathbf{x}), \tag{6.13}$$

where $\mathbf{W} \in \mathbb{R}^{p\times n}$ is the learnable parameter and $\sigma(\cdot)$ is a non-linearity or activation function. It is obvious that perceptron is a non-linear model and learning $\mathbf{W}$ for it leads to the non-linear ERM problem

$$\mathbf{W}^\star = \underset{\mathbf{W}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} l(\mathbf{y}_i, \sigma(\mathbf{W}\mathbf{x}_i)) + g(\mathbf{W}), \tag{6.14}$$

where $\sigma(\cdot)$ is point-wise non-linearity and $g(\cdot)$ is a regularizer.

The neural network or multi-layer perceptron can be built by concatenating perceptrons to create hidden layers for having a more complex and general family of functions. The latent variable in hidden layer l can be depicted in a recursive form as

$$\mathbf{z}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{z}^{(l-1)}), \tag{6.15}$$

where $\mathbf{z}^{(0)} = \mathbf{x}$ is called input layer, $\mathbf{z}^{(L)} = \mathbf{y}$ is called output layer, and the rest are hidden layers. Moreover, $\mathbf{W}^{(l)} \in \mathbb{R}^{p_l \times p_{l-1}}$ is the parameter matrix of l-th layer, where $p_0 = n$ is the input data dimension, $p_L = p$ is the output dimension. The equation for an L-layer neural network is

$$\mathbf{y} = \sigma\left(\mathbf{W}^{(L)}\sigma\left(\ldots\sigma\left(\mathbf{W}^{(2)}\sigma\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)\right)\right), \tag{6.16}$$

And we can collect model parameters in set $\mathcal{W} = \{\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L)}\}$, where L is the number of hidden layers in the network. Finally, the ERM problem for a neural network can be shown as

$$\mathcal{W}^\star = \underset{\mathcal{W}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} l\left(\mathbf{y}_i, \sigma\left(\mathbf{W}^{(L)}\sigma\left(\ldots\sigma\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)\right)\right) + g(\mathcal{W}), \tag{6.17}$$

One effective and understandable way to demonstrate the neural networks is nested cascade form as in Fig. 6.6.

Theoretically, based on the *universal function approximation theorem* [1], a fully-connected neural network (FCNN) can approximate any function. However the FCNN is a strong estimator model, when the dimension

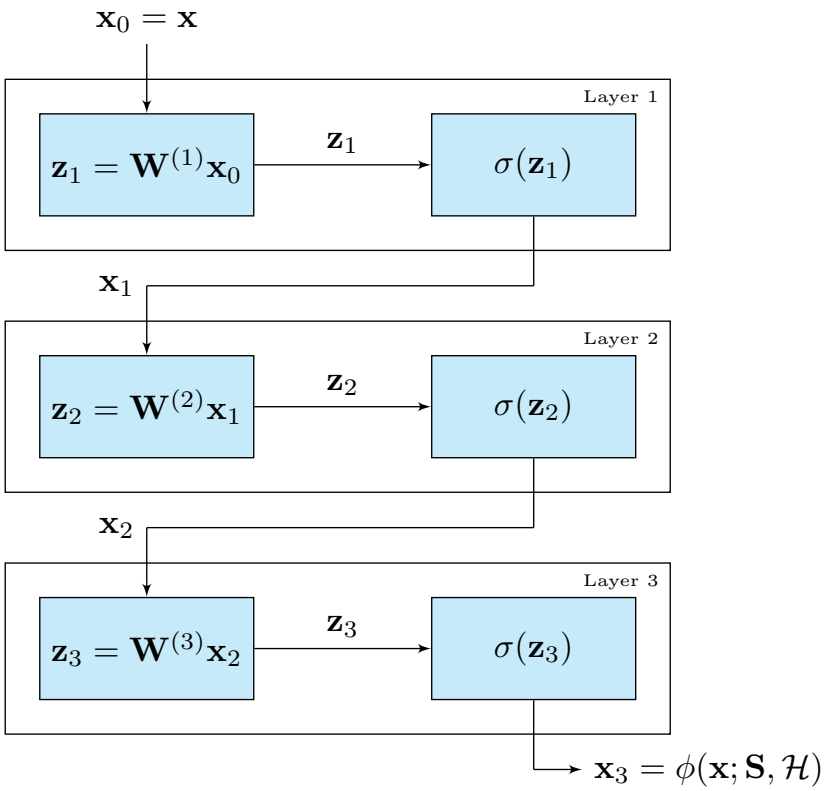


Figure 6.6: The neural network in nested cascade form with 3 layers. It is clear in each hidden layer a linear transformation and a non-linear activation function are applied on the previous layer's output. Also the input **x** and output **y** are in input layer and output layer respectively.

of input data increases, the number of model parameters grows significantly. In practice these models may have very high complexity and become computationally unaffordable. A solution to tackle the quest of dimensionallity is to add some restriction to the model based on a prior knowledge of the problem. The typical example are convolutional neural networks that exploit the adjacency of the points in space as a prior to reduce effectively the computational cost and number of parameters. Instead, when the data will be on the form of a graph we can use the adjacency of the points given by the graph to parameterize the FCNN with a graph prior, which will lead to graph neural networks.

6.5. Further Reading

The interested readers are encouraged to look up the following works: for statistical learning and ERM [4, 3, 5], for the connection between ERM and machine learning problems [6, 2], and for neural networks [1].

Bibliography

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

[2] A. Jung. "Explainable empirical risk minimization". In: *arXiv preprint arXiv:2009.01492* (2020).

[3] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[4] K.P. Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[5] S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[6] S. Sun et al. "A survey of optimization methods from a machine learning perspective". In: *IEEE transactions on cybernetics* 50.8 (2019), pp. 3668–3681.

7

Graph Neural Networks

7.1. Introduction

Fully-connected neural networks have two main issues: computational complexity of $\mathcal{O}(N^2)$ and number of parameters of $\mathcal{O}(N^2)$; so they cannot be used for large dimensions or when there is little data present. Hence, to address this problem, we need to extract structure in the data to act as an inductive bias [2]. Other issues of using neural networks for graphs are:

- if a new node attaches, we need to retrain the model parameters;
- if we retrain the model, it cannot be transferred into another very similar graph;
- we need to be careful of the node labelling and keep them in a look-up table.

There are other conventional models which use the structure of the data to process them more efficiently. Convolutional neural networks (CNNs) use spatial proximity for spatial data like images. Recurrent neural networks (RNNs) use temporal proximity for temporal data like time-series. In both of these models the linear component is substituted by a bank of filters exploiting the spatial and the temporal proximities.

For graph data, the graph is a natural inductive bias to be imposed as an inductive prior into the neural network. So, we will fix the class of functions to the one that considers the structure of the graph characterizing the data and use graph filters to perform the linear aggregating component.. The main goal of this chapter is to replace the fully connected layer in conventional neural networks with graph filters and show how they help defining graph neural networks.

7.2. Learning with Graph Filters

The graph filter is the central building block of graph convolutional neural networks (GCNNs.) Recall a graph filter is a linear system that uses graph convolution to process the input graph signal and provides another graph signal as the output. The in-out relation of a graph filter of order K is

$$\mathbf{y} = \mathbf{h}(\mathbf{S})\mathbf{x} = \sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x}. \tag{7.1}$$

where h_k are the scalar filter coefficients, $\mathbf{S}$ is the graph shift operator, and $\mathbf{h}(\cdot)$ is the matrix function of the graph filter. An illustration of this filter is showed in Fig. 7.1. Now, we can see the output of the graph filter as an embedded representation of the input signal that contains higher level information $\mathbf{y}$. In this way, learning parameters h_k allows learning the embedding $\mathbf{y}$ w.r.t. the input $\mathbf{x}$ and topology $\mathbf{S}$.

Assume we have a data set $\mathcal{T} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^{m}$ and the loss function $l(\cdot)$. Then, the parametric ERM problem for the graph filter in (7.1) can be written as

$$\mathbf{h}^\star = \underset{\mathbf{h}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{y}_i, \mathbf{h}(\mathbf{S})\mathbf{x}_i), \tag{7.2}$$

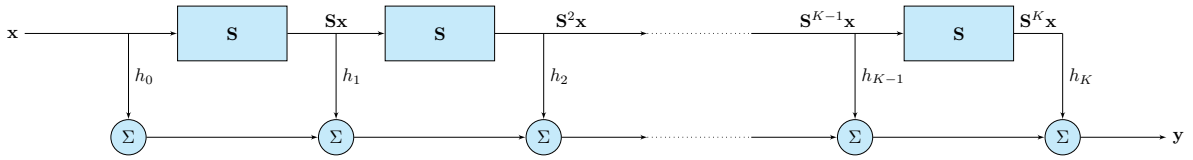


Figure 7.1: The block diagram of the graph filter.

where $\mathbf{h} = [h_0, \ldots, h_K]^\top$ is a vector containing the parameters in $\mathbf{h}(\mathbf{S})$. Notice that the graph filter has $K+1$ parameters and because of locality the computational complexity is $\mathcal{O}(M(K+1))$, where M is the number of the edges.

Sometimes, the desired output is not another graph signal or it has different dimension. In this case, we add a *readout* layer at the filter's output to match the dimension. The readout layer can be trainable or fixed. For example, it can be a trainable matrix that maps the graph filter's embeddings to the output, or it can be a fixed all-one vector to sum up the signal values at all nodes. The ERM problem for those scenario is

$$\mathbf{h}^\star = \underset{\mathbf{h}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{y}_i, \mathbf{A} \times \mathbf{h}(\mathbf{S})\mathbf{x}_i), \tag{7.3}$$

where $\mathbf{A}$ is the readout matrix. Fig. 7.2 illustrates a graph filter with readout layer in a block diagram form.

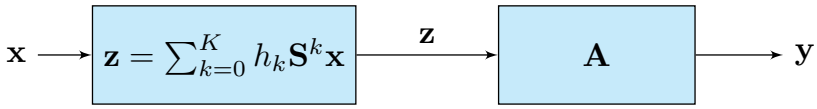


Figure 7.2: The block diagram of a graph filter with readout layer. $\mathbf{A}$ is the readout matrix.

All in all, graph filters are structured linear maps and they have limited expressive power. Hence, with analogy to neural networks, we will extend these linear maps to non-linear and nested architectures to learn different levels of representation; from low-level (input layer) to high-level (hidden layers).

7.3. Components of Graph Convolutional Neural Networks

In this section we will introduce the main elements of a graph neural network and use them in the following sections.

7.3.1. Graph perceptron

A graph perceptron is a graph filter followed by a point-wise nonlineaity and it is defined as

$$\mathbf{y} = \sigma\left[\sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x}\right], \tag{7.4}$$

where $\sigma[\cdot]$ is a point-wise nonlinear function such as ReLU, hyperbolic tangent, or absolute value. The block diagram of a graph perceptron is shown in Fig. 7.3.

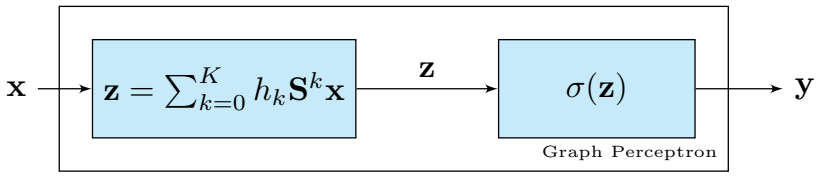


Figure 7.3: The block diagram of the graph perceptron.

By assuming the graph perceptron as a class of functions,we can learn a nonlinear map still defined by parameters h_k, which the ERM problem is of the form

$$\mathbf{h}^\star = \underset{\mathbf{h}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l\left(\mathbf{y}_i, \sigma\left[\sum_{k=0}^{K} h_k \mathbf{S}^k \mathbf{x}_i\right]\right). \tag{7.5}$$

Obviously, the graph perceptron model can learn non-linear maps and it is more expressive than a graph filter while upholding the locality. The number of parameters and computational complexity remain the same as for the graph filter.

7.3.2. Multi-layer graph perceptron

The graph perceptron learns only a low-level representation for the input graph signal; i.e., the non-linearized output of a graph perceptron is not further processed but it is considered as the final embedding However, this output can be seen in turn as another graph signal which can be processed with another graph perceptron to obtain a higher level representation. Hence, we consider a composite structure and stack L layers of graph perceptrons that form a multi-layer perceptron or a GCNN as illustrated in Fig. 7.4.

To be more specific, the output of first hidden layer is a latent graph signal as

$$\mathbf{x}_1 = \sigma[\mathbf{z}_1] = \sigma\left[\mathbf{h}(\mathbf{S})\mathbf{x}\right] = \sigma\left[\sum_{k=0}^{K} h_{1k}\mathbf{S}^k\mathbf{x}\right], \tag{7.6}$$

and in a same way, for the second layer we have

$$\mathbf{x}_2 = \sigma[\mathbf{z}_2] = \sigma\left[\mathbf{h}(\mathbf{S})\mathbf{z}_1\right] = \sigma\left[\sum_{k=0}^{K} h_{2k}\mathbf{S}^k\mathbf{z}_1\right]. \tag{7.7}$$

In general, for the output of l-th layer we have the following recursive equation

$$\mathbf{x}_l = \sigma[\mathbf{z}_l] = \sigma\left[\mathbf{h}(\mathbf{S})\mathbf{z}_{l-1}\right] = \sigma\left[\sum_{k=0}^{K} h_{lk}\mathbf{S}^k\mathbf{z}_{l-1}\right], \tag{7.8}$$

where $\mathbf{z}_l$ is the latent output of l-th layer, h_{lk} are the model parameters of l-th layer, and K is the filter order. Then, by using the recursive equation (7.8), we define the output function $\Phi(\cdot)$ of a multi-layer perceptron with L layers as

$$\mathbf{y} = \Phi(\mathbf{x};\, \mathbf{S}, \mathbf{h}_1, \ldots, \mathbf{h}_L) = \Phi(\mathbf{x};\, \mathbf{S}, \mathscr{H}), \tag{7.9}$$

where $\mathscr{H} = \{\mathbf{h}_1, \ldots, \mathbf{h}_L\}$ is the set of model parameters in all layers.

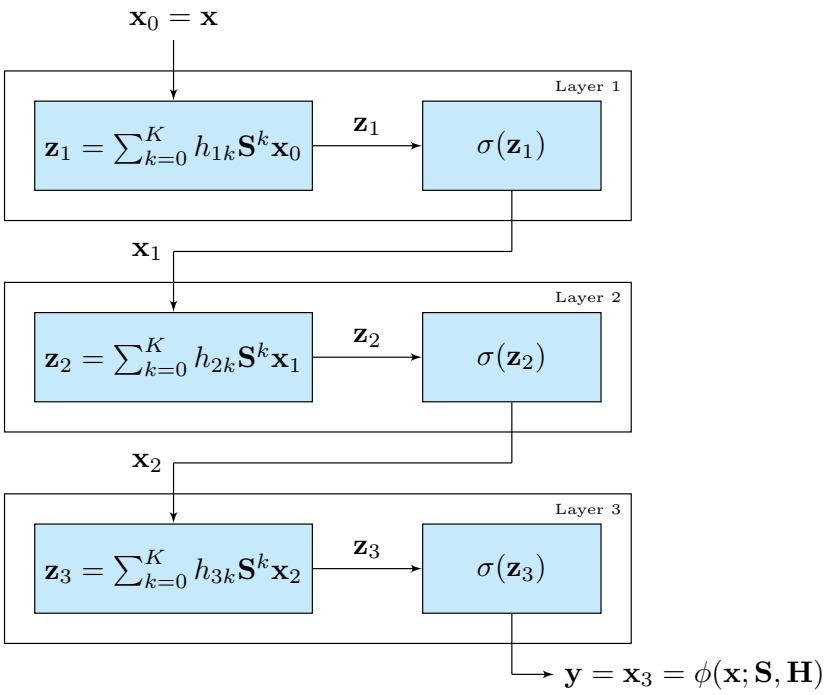


Figure 7.4: The block diagram of a multi-layer perceptron with $L = 3$.

The GCNN output is the output of the L-th graph perceptron, and its parametric ERM problem is

$$\mathscr{H}^{\star} = \underset{\mathscr{H}}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} l(\mathbf{y}_i, \Phi(\mathbf{x}_i; \mathbf{S}, \mathscr{H})). \tag{7.10}$$

The number of parameters for a L-layer GCNN is $L(K+1)$ and so the computational complexity is $\mathscr{O}(ML(K+1))$.

7.4. Graph Convolutional Neural Networks

The GCNN is the generalized form of multi-layer graph perceptrons, which instead of processing a feature signal per layer it processes multiple of them. Multi-layer perceptron uses only a single filter per layer. We can increase its expressive power using multiple graph filters in parallel in the form of a graph filter bank. This is similar to what is done in spatial CNNs.

To handle a graph filter with multiple features we need to define filter banks first. The filter bank is a collection of graph filters. For a filter bank composed of F filter, the output of f-th graph filter in the bank is

$$\mathbf{z}^f = \sum_{k=0}^{K} h_k^f \mathbf{S}^k \mathbf{x} \quad for \quad f = 1, \ldots, F. \tag{7.11}$$

The output of filter bank is the collection of F graph signals $\mathbf{z} = [\mathbf{z}_1, \ldots, \mathbf{z}_F]^\top$. The input of graph filter bank is a vector $\mathbf{x}$ that assigns a scalar on each node while the output is the matrix $\mathbf{z}$ which assigns a vector of dimension F on each node. Fig. 7.5 depicts the input-output relation of the graph filter bank with a block diagram.

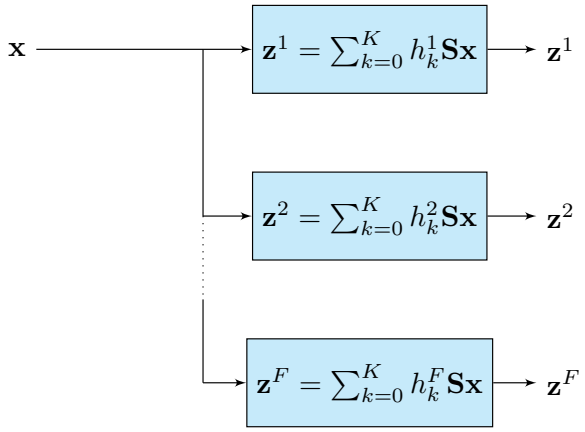


Figure 7.5: The block diagram of a graph filter bank with input dimension one and output dimension F.

We expect that each filter in the bank captures a specific part of the input signal and process it particularly. Altogether, the filter bank will increase our freedom to learn more powerful representations for the input signal $\mathbf{x}$. We can compactly write the input-output relation in matrix form

$$\mathbf{z} = \sum_{k=0}^{K} \mathbf{S}^k \mathbf{x} \mathbf{h}_k^\top, \tag{7.12}$$

where $\mathbf{h}_k = [h_k^1, \ldots, h_k^F]^\top$ is the vector of coefficients. Such equation shows how the input signal $\mathbf{x}$ is transformed into a collection of multiple signals $\mathbf{z}$.

We are also interested in processing of multiple-feature signals in the middle layers of a GCNN. Then we need to generalize graph filter banks for this kind of input. Consider there are F_0 input features $\{\mathbf{x}^1, \ldots, \mathbf{x}^{F_0}\}$ and F_1 output features $\{\mathbf{z}_1^1, \ldots, \mathbf{z}_1^{F_1}\}$ for the input layer. Then we need a graph filter bank of dimension $F_1 \times F_0$ to transform each input to a corresponding output. Each input graph signal $\mathbf{x}^g$ is processed with a bank of F_1 filters as

$$\mathbf{u}^{fg} = \mathbf{h}^{fg}(\mathbf{S})\mathbf{x}^g = \sum_{k=0}^{K} h_k^{fg} \mathbf{S}^k \mathbf{x}^f; \quad g = 1, \ldots, F_0 \quad f = 1, \ldots, F_1. \tag{7.13}$$

So in output we have $F_1 F_0$ graph signals, and if we keep repeating this again, we will go into an exponential scaling. To keep the number of filters limited, we sum the output over the same input index g to obtain the aggregated feature

$$\mathbf{z}^f = \sum_{g=1}^{F_0} \mathbf{u}^{fg} = \sum_{g=1}^{F_0} \sum_{k=0}^{K} h_k^{fg} \mathbf{S}^k \mathbf{x}^f; \quad f = 1, \ldots, F_1. \tag{7.14}$$

Likewise (7.12), we can write this equation in the compact matrix form

$$\mathbf{z} = \sum_{k=0}^{K} \mathbf{S}^k \mathbf{x} \mathbf{h}_k, \tag{7.15}$$

where $\mathbf{x} = [\mathbf{x}^1, \ldots, \mathbf{x}^{F_0}]$ is the collection of input signals and $\mathbf{h}$ is a $F_0 \times F_1$ matrix of coefficients collecting all scalars h_k^{fg} in (7.14).

With this in place, we can then build a GCNN with filter banks and more layers. The recursive equation for the l-th layer is

$$\mathbf{x}_l = \sigma[\mathbf{z}_l] = \sigma\left[\sum_{k=0}^{K} \mathbf{S}^k \mathbf{x}_{l-1} \mathbf{h}_{lk}\right]. \tag{7.16}$$

If we consider L layers for the GCNN, its parametric function will be

$$\mathbf{y} := \mathbf{x}_L = \Phi(\mathbf{x}; \mathbf{S}, \mathscr{H}), \tag{7.17}$$

where $\mathscr{H} = \{\mathbf{h}_l^{fg}\}_{lfg}$ is the set of trainable model parameters. An illustration of a GCNN is shown in Fig 7.6.

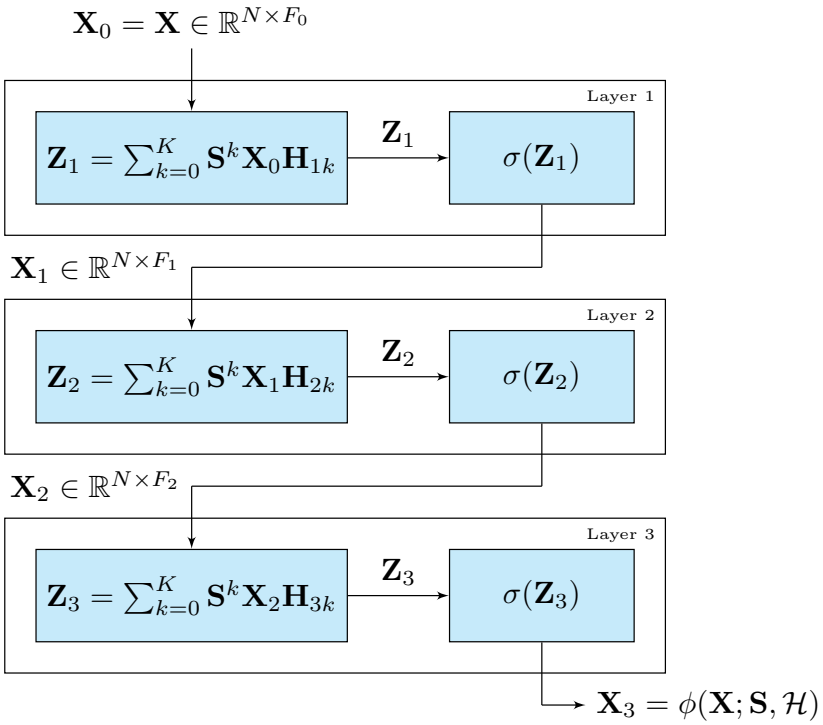


Figure 7.6: The block diagram of a GCNN with $L = 3$ layers.

7.5. Particular GCNN forms

In this section, we discuss particular forms of graph convolutional neural networks (GCNNs), elaborated in [17, 23, 25, 1]. In what follows, we focus on the neural network models in [17] and [23], and they are a subclass of the GCNN model discussed in Section 7.4. Recall the form of GCNN in the l-th layer with a convolutional filter of order K, i.e.,

$$\mathbf{X}_l = \sigma\Big[\sum_{k=0}^{K} \mathbf{S}^k \mathbf{X}_{l-1} \mathbf{H}_{lk}\Big] \tag{7.18}$$

where $\mathbf{X}_l$ and $\mathbf{X}_{l-1}$ are the output and input feature matrices in the l-th layer, $\mathbf{S}$ is the graph shifting matrix and $\mathbf{H}_{lk} \in \mathbb{R}^{F_{l-1} \times F_l}$ is the filter coefficient matrix with F_l being the number of features in the l-th layer.

7.5.1. Graph Convolutional Networks

The authors in [17] proposed the graph convolutional network (GCN) which has the following form in the l-th layer

$$\mathbf{X}_l = \sigma[\mathbf{S}\mathbf{X}_{l-1}\mathbf{H}_l] \tag{7.19}$$

where in [17] the shift operator $\mathbf{S}$ is specified to be the adjacency matrix $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}}$ with $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ containing self-connections, and $\tilde{D}_{ii} = \sum_{j \in \mathcal{N}_i} \tilde{A}_{ij}$, and $\mathbf{H}_l \in \mathbb{R}^{F_{l-1} \times F_l}$ is the trainable linear transformation matrix. The graph convolutional part in (7.19), i.e., $\mathbf{S}\mathbf{X}_{l-1}\mathbf{H}_l$ considers only one-hop shifting for each layer and filter, i.e., $K = 1$ and $\mathbf{H}_0 = \mathbf{0}$ for all layers in GCNN [cf.(7.18)]. However, notice that since $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$, we may also see the convolution part in (7.19) as $\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{A}\tilde{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}_{l-1}\mathbf{H}_l + \tilde{\mathbf{D}}^{-1}\mathbf{X}_{l-1}\mathbf{H}_l$. This implies that setting $\mathbf{S} = \mathbf{A}$, we have the same filter coefficients $\mathbf{H}_l$ for both $k = 1$ and $k = 0$. The latter interpretation depends largely on how we fix the shift operator. In both cases, the GCN model (7.19) is a particular case of the GCNN model in (7.18).

With this particularization, the GCN has the following properties:

- it is easier to understand, because in each layer, each node exchanges information with their direct neighbors once;
- the feature propagation rule has a faster implementation, because there is only one feature shifting and a nonlinearity per layer;
- the number of parameters in (7.18) is reduced by approximately K times in (7.19), because the GCN only has one parameter matrix at each layer.

However, due to its simplicity, the convolution in GCN is highly limited to model simple functions. For the case that $\mathbf{S}$ is specified as $\hat{\mathbf{A}}$, by performing the graph Fourier transform with the eigenvectors of the normalized graph Laplacian $\tilde{\mathbf{L}} = \mathbf{I}_N - \hat{\mathbf{A}}$, we have the spectral response of $\hat{\mathbf{A}}\mathbf{X}$ as $(1 - \tilde{\lambda}_i)$ at graph frequency $\hat{\lambda}_i$, which is the i-th eigenvalue of $\tilde{\mathbf{L}}$.

7.5.2. Simplifying Graph Convolutional Networks (SGCs)

Simplifying graph convolutional network (SGC) considers first shifting the signal L times (as the number of layers) and applying the final nonlinearity to the output [23]. With an input feature $\mathbf{X}_0$, an L-layer SGC has the following output-input form

$$\mathbf{X}_L = \sigma[\mathbf{S}^L\mathbf{X}\mathbf{H}] \tag{7.20}$$

which considers (7.19) with only the L-hop filter tap, i.e., $\mathbf{H} = \mathbf{0}$ for all $k < L$. So, L layers of SGC equals to a single layer GCNN with $K = L$, which reflects the low descriptive power of the SGCs. The shifting matrix $\mathbf{S}$ is specified to be $\hat{\mathbf{A}}$, same as in GCN [23]. From (7.20) we can see the SGC as stacking L layers of GCN in (7.19) and removing the nonlinearities, but this results to be a single layer GCNN with only the highest power $K = L$ coefficient being nonzero.

The SGC has a fast implementation as well, because the final SGC output can be obtained by L times graph shifting and one nonlinearity, and the number of parameters is the same as the GCN. However, the SGC is limited to the higher-order polynomial functions, because the spectral response of the convolutional operation $\hat{\mathbf{A}}^L\mathbf{X}$ is $(1 - \tilde{\lambda}_i)^L$ at graph frequency $\tilde{\lambda}_i$.

Remark. The goal of both GCN (7.19) and SGC (7.20) is to avoid the sum of K feature shifting operations in the GCNN (7.18), so a faster implementation can be done. However, their expressive power is limited. They can be useful in problems where information on the data structure is available [17, 23], because their particularization can be seen as a regularization of the graph convolution, constraining the representation space to be a subspace of that in the GCNN model. If a better performance is needed, the GCNN should then be considered as a viable solution because it has higher learning capabilities.

7.6. Graph pooling

Here we will discuss different graph pooling methods shortly for the sake of completeness. Interested readers can refer to [24] for more details. A pooling layer can be added before nonlinearities same as in convolutional neural networks, see Fig. 7.7. The idea of pooling is to reduce the graph into a smaller one, so that the

computation of convolutions have cheaper cost and complexity. There are three categories of graph pooling modules: node selection; clustering; zero-padding.

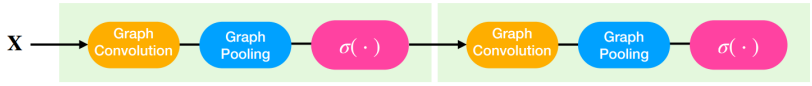


Figure 7.7: Graph pooling layer in a GCNN.

7.6.1. Node selection pooling.

Node selection pooling downsamples the graph by selecting the most important nodes as shown in Fig. 7.8. To this end, an importance measure is assigned to all of the nodes which considers the data over the graph. Then it ranks the nodes based on their importance and selects top N_p nodes. So, the other nodes will be eliminated from the adjacency matrix and the graph signal over selected node will pass through a nonlinearity. Hence, the edges in the new graph and the graph signal remain the same, only less important nodes are eliminated with their corresponding edges and graph signal. The main advantage of this method is its very fast implementation. One of the pooling algorithms in this category is *gPool* [10].

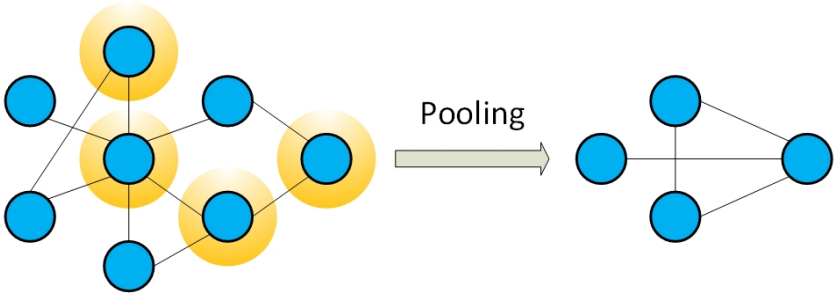


Figure 7.8: Node selection algorithm for graph pooling.

7.6.2. Clustering pooling

Clustering pooling methods downsample the graph by clustering the nodes into N_p clusters and assuming each of them as a new node as illustrated in Fig. 7.9. The clustering can be done by another GNN like in *DiffPool* [26], or based on the graph Eigen decomposition as in *EigenPooling* [18]. The graph signal is reduced over each cluster by mean/max pooling and a nonlinearity. In summary, the new graph assumes each cluster as a node, these nodes have an edge if the clusters were connected, and the graph signal is obtained by a summarization function over each cluster. The clustering pooling algorithms consider both the data and the graph topology but they are not efficient in implementation.

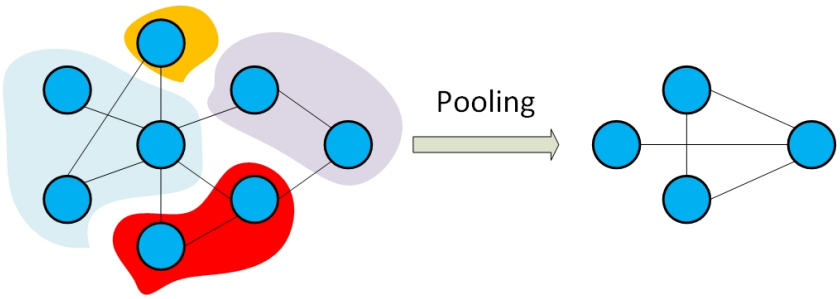


Figure 7.9: Clustering pooling method for graph pooling.

7.6.3. Zero-pad pooling

Unlike the other approaches, zero-pad pooling keeps the graph structure. The idea behind this technique is to set some signals over the graph to zero, so that we can use the sparsity and reduce the computational cost

and complexity at the next layer [7], see Fig. 7.10. Zero-pad pooling reduces the number of the features of the graph signal to reduce computational cost. It has two main steps:

- Downsample: combining the features using a sampling matrix to reduce the number of active features.
- Summarization: generating the output graph signal by applying mean/max function over the combined features.

So, the output graph signal has fewer active features and then the next layer tolerates less computational complexity.

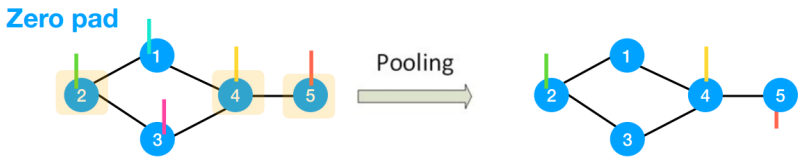


Figure 7.10: An illustration of zero-pad pooling in GCNNs.

7.6.4. Global aggregation

The global aggregation layer is often used in the output layer to aggregate all node features into a global vector of features as shown in Fig. 7.11. This aggregation can be done by summarization functions such as mean or max. So, the global aggregation layer accepts graphs of different sizes and map their embeddings into a fixed-size representation feature vector [3]. Due to this property, the global aggregation layer is usually applied for graph classification or any other graph-level related task.

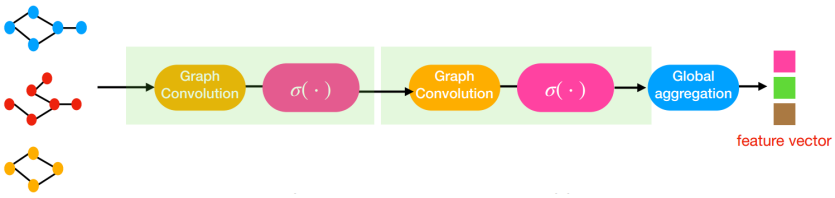


Figure 7.11: A GCNN with a global aggregation layer at the output.

7.7. Properties of GCNNs

In this section, we discuss some important properties of GCNNs.

7.7.1. Complexity and parameters

The essential component inside the GCNN is the graph filter, which governs both the number of learnable parameters and computational complexity. A graph filter has $K+1$ parameters and since the graph convolution is a local operator over the graph, the computational complexity is $\mathcal{O}(MK)$ where M is the number of graph edges. A graph filter bank in a GCNN contains F^2 graph filters, and thus, it has KF^2 parameters. Again, because of the locality of graph convolution, the computational complexity is $\mathcal{O}(MKF^2)$. Now, a GCNN is the stacked version of L graph filter banks and has KF^2L parameters. So, the computational complexity of a GCNN is $\mathcal{O}(MKF^2L)$.

There are two important remarks here. First, the number of parameters for a GCNN is independent from the number of graph nodes and makes the GCNN able for processing large graphs. Second, the computational complexity of GCNN is linear in the number of edges M.

7.7.2. Training and standard tasks

The GCNN can be trained with stochastic gradient descent algorithm, and since it has stacked structure, we use back propagation to train its parameters. The readout layer should be designed to map the GCNN output into our desired task. We illustrate a few here.

Classifying graph signals

Suppose that we want to classify graph signals into two categories, say normal and abnormal. This is for instance the case when a network of sensors measures a physical signal, e.g., temperature. In this case, we stack the output of the GCNN into a vector and add a fully connected (FC) layer to map from the dimension of the GCNN output to the binary classes. To keep the number of parameters limited, we can share the parameters of the final FC layer among nodes. This final output goes to the hit loss function, which will be minimized by back propagation to learn the GCNN parameter. Fig. 7.12 illustrates the graph signal classification task.

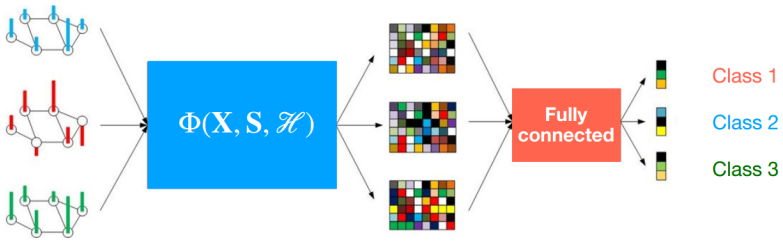


Figure 7.12: Graph signal classification with the GCNN.

Regression over signals

Here, we intend to perform a regression over graph signals. This tasks consist of inferring a single scalar from the node features and the topological information. An application of the latter is for instance in rating prediction [8]. Applications to distributed processing over sensor networks are again a potential scenario. Again, we use a fully connected layer at the output of the GCNN to map the final GCNN output into the scalar value. But we need to share the parameters of fully connected layer over the nodes to keep the GCNN distributed. So the output of i-th node will be

$$y_i = \mathbf{h}_{\mathrm{FC}}^{\top} \mathbf{x}_{Li}, \tag{7.21}$$

where $\mathbf{h}_{FC}$ is the vector of shared parameters in the fully connected layer. The loss function is usually minimum mean squares (MSE) and can be minimized by back propagation. Fig. 7.13 shows a graph signal regression task with shared parameters in final FC layer. For more details, interested readers can refer to [15].

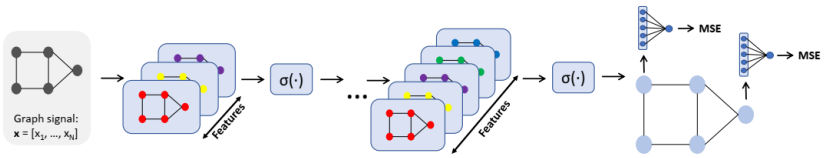


Figure 7.13: Graph signal regression with the GCNN. The Fc layer parameters are shared among nodes.

Node classification

The goal is to classify nodes of a graph into classes based on both graph topology and available data. An application of the latter is in the popular CORA dataset which explained in Section 7.9.1. This is a semi-supervised task since we have only a small portion of nodes' labels and should infer the labels for the remaining nodes. To this end, the embeddings of each node passes through a FC layer which has the labels on its output. The loss function is often an entropy loss and it should be minimized over the nodes with pre-known label. Fig. 7.14 illustrated the node classification task using a GCNN model. The loss function is minimized by back propagation over the nodes with pre-known label, and the trained GCNN will be applied to all nodes again to predict their labels. For more details refer to [17].

Graph classification

The goal is to classify different graphs based on their topological structure and the data over them. So, in this task, we have distinct graphs and different graph signals as input. To obtain a compact representation on

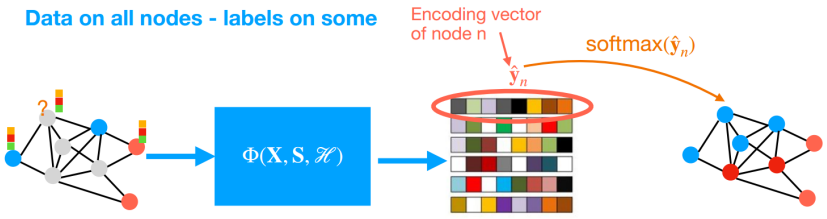


Figure 7.14: Node classification task by the GCNN model.

the graph level, GCNNs are often combined with pooling and readout layers. So the output of GCNN should be cascaded by a aggregator to generate the classified output. Again, the network can be trained with back propagation over a loss function at the output. Fig. 7.15 illustrated the graph classification task by a GCNN model. Refer to [6, 25] for more details.

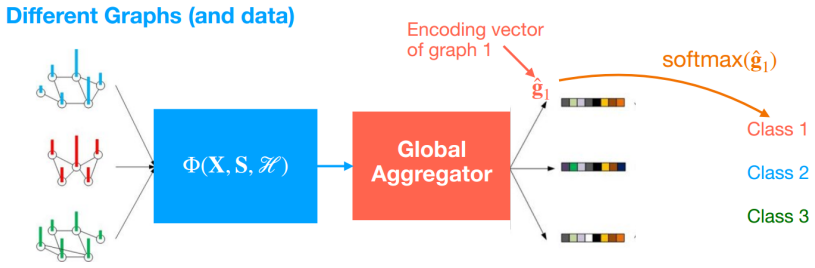


Figure 7.15: Graph classification task by the GCNN model.

7.7.3. Distributed aspects

Assume a GCNN without pooling layers. All the components of a GCNN are distributable: the graph filter are local operators and can be distributed over the neighborhood of each nodes; and the non-linearities are point-wise. So we can tell this property is inherited directly from graph filters. Hence, we just now need to run multiple filters in parallel.

7.7.4. Permutation equivariance

As we observed in equation (7.9), the output of GCNN depends on both the graph $\mathbf{S}$ and model parameters $\mathscr{H}$. Hence, once the GCNN is trained and model parameters $\mathscr{H}$ are obtained, we can treat the graph $\mathbf{S}$ just as a support. So, if we have a new another graph $\tilde{\mathbf{S}}$, we can just apply the same learned coefficients $\mathscr{H}$ to the new inputs without any change. This is because the filter parameters h_k are similar for all nodes irrespectively on the graph dimensions. In other words, a trained GCNN implies a set of coefficients which are graph agnostic and can be deployed everywhere. This property is particularly useful as it allows training GCNNs on a graph $\mathscr{G}$ and deploying them on a permuted version on it $\mathscr{G}'$ without bookkeeping the labels. This property, known as *permutation equivariance*, is stated in the following proposition.

Proposition: Let $\mathbf{x}$ be a graph signal defined on the vertices of a graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ with shift operator $\mathbf{S}$. Consider also the output of a GCNN $\Phi(\mathbf{x}; \mathbf{S})$. Then, for a permutation matrix $\mathbf{P}$, it holds that

$$\mathbf{P}^\top \Phi(\mathbf{x}; \mathbf{S}) = \Phi(\mathbf{P}^\top \mathbf{x}; \mathbf{P}^\top \mathbf{S}\mathbf{P}). \tag{7.22}$$

That is, the GCNN output operating on the graph $\mathscr{G}$ with input $\mathbf{x}$ is a permuted version of the GCNN output operating on the permuted graph $\mathscr{G}' = (\mathscr{V}', \mathscr{E}')$ with permuted shift operator $\mathbf{S}' = \mathbf{P}^\top \mathbf{S}\mathbf{P}$ and permuted input signal $\mathbf{x}' = \mathbf{P}^\top \mathbf{x}$.

This proposition establishes the output of a GCNN is independent of node labeling. This is important since it explains how the GCNN can extract topological symmetries of the underlying graph to process the signals.

7.8. Other Graph Neural Network Forms

As we saw in previous sections, graph neural networks (GNNs) consist of a concatenation of layers, in which each layer applies a *graph filter* followed by a pointwise nonlinearity. We can essentially view GNNs as a minor modification of graph filters, i.e., graph filters and pointwise nonlinearities. This minor modification allows to learn nonlinear mappings between input and output representations.

If we consider different types of graph filters in the GNN layers, we can expect different graph neural network forms. In Chapter 9, we have introduced different types of graph filters, for instance, node and edge varying graph filters [20, 4, 16]. Thus, we will first briefly introduce the alternative GNN forms based on such graph filters. Then, we will discuss the *graph attention networks (GATs)*, which deal with the cases where the graph edge weights are unknown and thus are able to learn them from data. Finally, we will present the *message passing neural networks (MPNNs)*, in which we build GNNs based on the idea of message exchanges between neighboring nodes.

7.8.1. EdgeNets: Edge Varying Graph Neural Networks

Graph convolution neural networks build upon the finite impulse response (FIR) graph filters. In this section, we build three variants of GNN by replacing the FIR graph filters with other variants. Note that we will stay on a schematic level and convey the main ideas and properties of these GNN variants. For a more detailed discussion, refer to [16].

Edge varying GNNs. Recall the definition of the edge varying graph filter in Chapter 9

$$\mathbf{H}(\mathbf{S}) = \Phi^{(0)} + \Phi^{(1)}\Phi^{(0)} + \cdots + \Phi^{(K)}\Phi^{(K-1)}\ldots\Phi^{(0)} \triangleq \sum_{k=0}^{K} \Phi^{(k:0)} \tag{7.23}$$

where $\Phi^{(k:0)} = \prod_{k'=0}^{k} \Phi^{(k')} = \Phi^{(k)}\Phi^{(k-1)}\ldots\Phi^{(0)}$ and $\Phi^{(k')} \in \mathbb{R}^{N\times N}$ for all $k' = 0,\ldots,K$, are edge varying weighting matrices with the same support as $\mathbf{S}+\mathbf{I}$. Compared to conventional graph filters with shifting matrix $\mathbf{S}$, this filter enjoys more degrees of freedom as the k-th order shifting matrix $\Phi^{(k:0)}$ is parametrized for each edge.

We can then define the edge varying GNN (EdgeNet) by composing the edge varying graph filter with a pointwise nonlinearity. Formally, consider a set of L layers $l = 1,\ldots,L$ and let $\mathbf{H}_l(\mathbf{S}) = \sum_{k=0}^{K} \Phi_l^{(k:0)}$ be the filter at layer l. Thus, an edge varying GNN at layer l has the form as

$$\mathbf{x}_l = \sigma[\mathbf{H}_l(\mathbf{S})\mathbf{x}_{l-1}] = \sigma\Big[\sum_{k=0}^{K} \Phi_l^{(k:0)}\mathbf{x}_{l-1}\Big] \tag{7.24}$$

where the initial input is $\mathbf{x}_0 = \mathbf{x}$ and the output is $\mathbf{x}_L$. We illustrate the edge varying shifting in Figure 7.16.

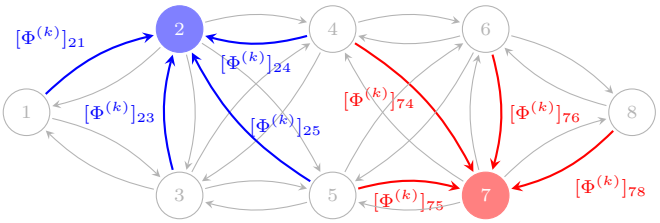


Figure 7.16: Edge varying shifting in EdgeNets where node 2 and node 7 are highlighted.

To augment the representation power of GNNs it is customary to add multiple node features per layer. We define matrices $\mathbf{X}_l = [\mathbf{x}_l^1,\ldots,\mathbf{x}_l^{F_l}] \in \mathbb{R}^{N\times F_l}$ with each column $\mathbf{x}_l^f$ representing a graph signal feature at layer l. These so-called features are cascaded through layers where they are processed by the composition of edge varying graph filters and pointwise nonlinearities according to

$$\mathbf{X}_l = \sigma\Big[\sum_{k=0}^{K} \Phi_l^{(k:0)}\mathbf{X}_{l-1}\mathbf{H}_{lk}\Big] \tag{7.25}$$

where $\mathbf{H}_{lk} \in \mathbb{R}^{F_{l-1}\times F_l}$ is a matrix of coefficients that affords flexibility to process different features with different filter coefficients. At layer l, (7.25) represents a bank of edge varying graph filters applied to a set of F_{l-1} input features $\mathbf{x}_{l-1}^g$ to produce a set of F_l output features $\mathbf{x}_l^f$.

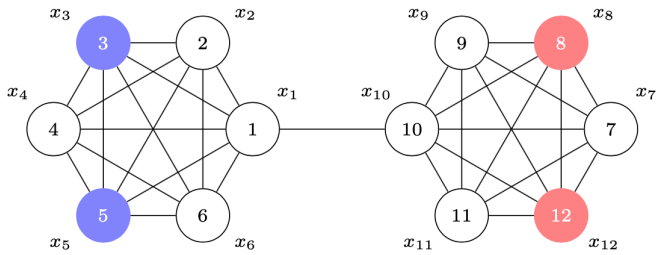


Figure 7.17: Permutation equivariance of machine learning on graphs. Many tasks in machine learning on graphs are equivariant to permutations but not all. For example, we expect agents 3, 5, 8, and 12 to be interchangeable from the perspective of predicting product ratings from the ratings of other nodes. But from the perspective of community classification we expect 3 and 4 or 8 and 12 to be interchangeable, but 3 and 5 are not interchangeable with 8 and 12.

Remark. [Permutation Equivariance] An important property of the graph convolution neural networks (GCNNs) is the permutation equivariance, but this property does not hold for the EdgeNet, because of the edge weight parameterization. However, this is not necessarily unappealing. For instance, permutation equivariance holds in recommendation systems but does not hold in community classification, as illustrated in Figure 7.17. In cases where equivariance is not a property of the task, GCNNs are not expected to do well. In GCNNs the filter forces all nodes to weight the information of all k-hop neighbors with the same parameter irrespectively of the relative importance of different nodes and different edges, which giving rise to the loss of local detail. However, we can use an EdgeNet that relies on the edge varying graph filter which would have each node i learn a different edge weight parameter $\Phi_{ij}^{(k)}$ for each neighbor j.

Remark. [Properties] As discussed in Chapter 9, there exists a distributed implementation of the edge varying graph filters, which holds true for EdgeNets as well. Furthermore, the edge varying graph filter is characterized by the $K+1$ parameter matrices and contains $K(M+N)+N$ parameters since these matrices admit the graph support and enjoys the sparsity of the graph shifting operator. Thus, by considering the multi-input and -output features in EdgeNets, the number of parameters per layer is at most $(K(M+N)+N)F^2$ where $F = \max_l F_l$. Likewise, the computational complexity at each layer is of order $\mathcal{O}(K(M+N)F^2)$.

Hybrid GNNs. GCNNs and EdgeNets can deal with cases where permutation equivariance is necessary and unnecessary, respectively. In addition, compared to GCNNs, there are more parameters to be learned in EdgeNets, which may often be too demanding. The appealing intermediate solution in both cases is to use filters with *controlled edge variability* to mix the advantages of a permutation equivariant GCNN with the processing of local detail by an EdgeNet.

We can make use of the *hybrid* filters introduced in Chapter 9, which are linear combinations of FIR graph filters and edge varying graph filters that operate on a subset of nodes. The formal definition has the form

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{K} \Big(\prod_{k'=0}^{k} \Phi_{\mathcal{I}}^{(k')} + h_k \mathbf{S}^k \Big) \tag{7.26}$$

where $\mathcal{I} \subset \mathcal{V}$ denotes an importance subset of $|\mathcal{I}|$ nodes applying local edge varying shifting matrices $\Phi_{\mathcal{I}}^{(k')}$ with entries $[\Phi_{\mathcal{I}}^{(k')}]_{ij} = 0$ for all $i \notin \mathcal{I}$ or $(i,j) \notin \mathcal{E}$. That is, the parameterized shifting matrices $\Phi_{\mathcal{I}}^{(k')}$ contains nonzero elements only at rows i that belong to set $\mathcal{I}$ and with node j being a neighbor of i. By replacing the edge varying graph filters with the hybrid filters in (7.26), we obtain a hybrid GNN.

$$\mathbf{X}_l = \sigma \Big[\sum_{k=0}^{K} \Big(\prod_{k'=0}^{k} \Phi_{\mathcal{I}}^{(k')} + \mathbf{S}^k \Big) \mathbf{X}_{l-1} \mathbf{H}_{lk} \Big] \tag{7.27}$$

In essence, nodes $i \in \mathcal{I}$ learn edge dependent parameters which may also be different at different nodes, while nodes $i \notin \mathcal{I}$ learn global parameters, see Figure 7.18 for an illustration of parameter sharing in hybrid GNN.

Remark. The number of parameters in hybrid filters depends on the total number of neighbors of all nodes in the importance set $\mathcal{I}$. Define then $M_{\mathcal{I}} = \sum_{i \in \mathcal{I}} N_i$ as the total number of edges to apply edge-dependent parameters and with N_i being the number of neighbors of node i. We then have $KM_{\mathcal{I}} + |\mathcal{I}|$ parameters in the edge varying filters and $K+1$ parameters in the conventional FIR filters. Therefore, we have a total of $(|\mathcal{I}| + KM_{\mathcal{I}} + K + 1)F^2$ parameters per layer in a hybrid GNN. Similarly, the hybrid GNN also admits a

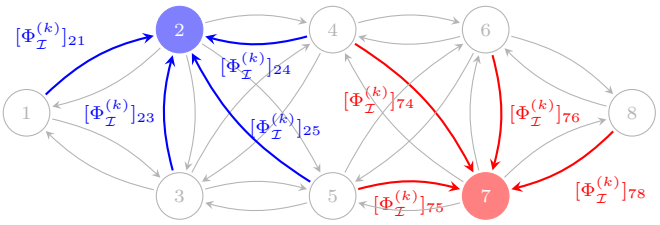


Figure 7.18: Parameter illustration in the shifting in Hybrid GNN, where node 2 and node 7 are in the importance node set which applies the local edge varying shifting matrix $\Phi^{(k)}$ and the rest nodes apply the global shifting parameter $\mathbf{S}$.

distributed implementation. The implementation cost of a hybrid GNN layer is of order $\mathcal{O}(KF^2(M+N))$ since both terms in (7.26) respect the sparsity of the graph.

Block GNNs. An alternative to hybrid graph filters to compromise between the permutation equivariance and the local detail is *block varying* graph filters. These filters use different coefficients in different parts of the graph. Formally, let $\mathcal{B} = \{\mathcal{B}_1, \ldots, \mathcal{B}_B\}$ be a partition of the node set into B blocks with block $\mathcal{B}_i$ having B_i nodes. Define the tall matrix $\mathbf{C}_{\mathcal{B}} \in \{0,1\}^{N \times B}$ such that $[\mathbf{C}_{\mathcal{B}}]_{ij} = 1$ if node i belongs to block $\mathcal{B}_j$ and 0 otherwise. Let also $\mathbf{h}_{\mathcal{B}}^{(k)} \in \mathbb{R}^B$ be a vector of block coefficients of filter order k. Block varying graph filters have the form

$$\mathbf{H}(\mathbf{S}) = \sum_{k=0}^{K} \operatorname{diag}(\mathbf{C}_{\mathcal{B}} \mathbf{h}_{\mathcal{B}}^{(k)}) \mathbf{S}^k \tag{7.28}$$

which assigns coefficient $[\mathbf{h}_{\mathcal{B}}^{(k)}]_i$ to all nodes in the subset $\mathcal{B}_i$. It can be considered as a special case of the node varying graph filters [20]. Replacing (7.24) with the block varying graph filters (7.28), it leads to a block varying GNN.

Remark. As block varying GNNs apply a bock varying filter which contains B blocks per layer for each output-input feature pair, for each block, there are $K+1$ coefficients. Thus, they have $B(K+1)F^2$ parameters per layer. At each layer, to apply the block varying filter, there needs a computational complexity of order $\mathcal{O}(KF^2M)$, as implementing each output-input feature takes a complexity of order $\mathcal{O}(KM)$. They can be implemented in a distributed fashion as well as the node varying graph filters.

7.8.2. Graph Attention Networks

Graph convolutions are efficient, enjoy distributed implementations and can also be analyzed theoretically. But they require the underlying graph to be well established and represent the data effectively to act as a powerful inductive prior. In some cases, the edge weights cannot be accessed or are known with some uncertainty and we need an architecture that learns from the data also the edge weights. EdgeNet is one choice but it may have again too many degrees of freedom.

Graph attention networks (GATs) is an alternative choice in this setting in which the number of parameters is independent of the graph dimensions [22]. In GATs, nodes learn edge weights for the task at hand directly from the features. We assume the input to the network is a node feature matrix $\mathbf{X}_0 = \mathbf{X} \in \mathbb{R}^{N \times F_0}$ and the input to the l-th layer is $\mathbf{X}_l \in \mathbb{R}^{N \times F_{l-1}}$. At the l-th layer, we first perform *self-attention* on the nodes, a shared attentional mechanism $a : \mathbb{R}^{F_{l-1}} \times \mathbb{R}^{F_{l-1}} \to \mathbb{R}$ computes *attention coefficients*

$$e_{ij} = a(\mathbf{H}\mathbf{x}_i, \mathbf{H}\mathbf{x}_j) \tag{7.29}$$

that indicate the *importance* of node j's features to node i, where $\mathbf{x}_i$ is the i-th row vector of the input feature matrix $\mathbf{X}_{l-1}$ and $\mathbf{H} = \mathbf{H}_l \in \mathbb{R}^{F_l \times F_{l-1}}$ is the linear transformation coefficient matrix at layer l. Figure illustrates the attention coefficient e_{13} based on the features of node 1 and 3.. Note that we omit the layer index l to simplify notations.

In its most general formulation, the model allows every node to attend on every other node, dropping all structural information. We instead preserve the graph structure (support) by performing the attention, i.e., we only compute e_{ij} for nodes $j \in \mathcal{N}_i$, the one-hop neighbors of node i in the graph. To make attention coefficients easily comparable across different nodes, we normalize them across all choices of j using the softmax function

$$\alpha_{ij} = \operatorname{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i \cup i} \exp(e_{ik})}. \tag{7.30}$$

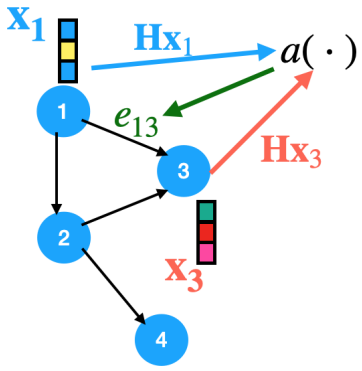


Figure 7.19: Attention coefficient computation illustration through mechanism (7.29), where the attention coefficient e_{13} is computed based on the features of node 1 and node 3.

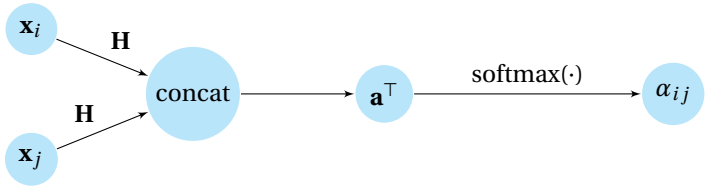


Figure 7.20: Schematic illustration of the attention mechanism (7.31).

In the original work [22], the attention mechanism $a(\cdot)$ is a neural network parameterized by a weight vector $\mathbf{a} \in \mathbb{R}^{2F_l}$, and applies the pointwise nonlinearity $\sigma[\cdot]$. The attention coefficients can be expressed as

$$\alpha_{ij} = \sigma\Big[\frac{\exp(\mathbf{a}^\top[\mathbf{H}\mathbf{x}_i \| \mathbf{H}\mathbf{x}_j]^\top)}{\sum_{k\in\mathcal{N}_i\cup i}\exp(\mathbf{a}^\top[\mathbf{H}\mathbf{x}_i \| \mathbf{H}\mathbf{x}_k]^\top)}\Big] \tag{7.31}$$

for all edges $(i,j) \in \mathcal{E}$, where $\|$ is the concatenation operation. This operation is illustrated in Figure 7.20.

Once we obtain the normalized attention coefficients, we can perform the graph convolutions and pointwise nonlinear mapping $\sigma(\cdot)$ on the input features $\mathbf{X}_{l-1}$ to obtain the output at node i at layer l as

$$[\mathbf{X}_l]_i = \sigma\Big[\sum_{j\in\mathcal{N}_i\cup i}\alpha_{ij}[\mathbf{X}_{l-1}\mathbf{H}]_j\Big]. \tag{7.32}$$

If we represent the attentional graph shifting matrix Φ with entries $[\Phi]_{ij} = \alpha_{ij}$, then the output feature $\mathbf{X}_l$ at layer l can be expressed as

$$\mathbf{X}_l = \sigma[\Phi\mathbf{X}_{l-1}\mathbf{H}]. \tag{7.33}$$

It has the similar form as the GCNN with one order graph filters $K = 1$, but in GATs, the shifting matrix Φ is learned from the features $\mathbf{X}_{l-1}$ that are passed from the layer $l-1$ following the above attention mechanism.

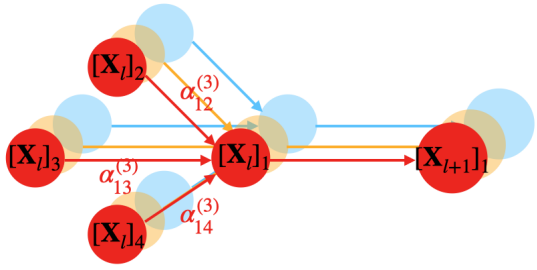


Figure 7.21: Multi-head attention illustration of node 1 [cf. (7.34) and (7.35)] [22], where different colors indicate different attention mechanisms.

To stabilize the learning process of the self-attention, [22] has further considered to employ *multi-head attention*. Specifically, P independent attention mechanisms execute the transformation of (7.32), and then their features are concatenated, resulting in the following output feature representation

$$[\mathbf{X}_l]_i = \Big\|_{p=1}^{P} \sigma\Big[\sum_{j\in\mathcal{N}_i\cup i} \alpha_{ij}^{(p)} [\mathbf{X}_{l-1}\mathbf{H}^{(p)}]_j\Big] \quad \text{and equivalently,} \quad \mathbf{X}_l = \Big\|_{p=1}^{P} \sigma\Big[\Phi^{(p)}\mathbf{X}_{l-1}\mathbf{H}^{(p)}\Big] \tag{7.34}$$

where $\|$ represents concatenation in the column direction, $\alpha_{ij}^{(p)}$ are normalized attention coefficients computed by the p-th attention mechanism $a^{(p)}(\cdot)$, and $\mathbf{H}^{(p)} = \mathbf{H}_l^{(p)}$ is the corresponding p-th linear transformation coefficient matrix at layer l. We again omit the layer index l for notation simplicity. Note that, in this setting, the final returned output at each layer, $[\mathbf{X}_l]_i$, will consist of PF_l features (rather than F_l) for each node due to the concatenation.

In the final layer of the network, concatenation is no longer sensible. Instead, it is employed with *averaging*, and delay applying the finally nonlinearity until then, which results in the following output [22]

$$[\mathbf{X}_l]_i = \sigma\Big[\frac{1}{P}\sum_{p=1}^{P}\sum_{j\in\mathcal{N}_i\cup i} \alpha_{ij}^{(p)} [\mathbf{X}_{l-1}\mathbf{H}^{(p)}]_j\Big]. \tag{7.35}$$

The total output multi-feature $\mathbf{X}_l$ can be expressed as

$$\mathbf{X}_l = \sigma\Big[\frac{1}{P}\sum_{p=1}^{P}\Phi^{(p)}\mathbf{X}_{l-1}\mathbf{H}^{(p)}\Big] \tag{7.36}$$

where $\Phi^{(p)}$ is the p-th self-attentional shifting matrix which shares the graph support. In Figure 7.21, the multi-head attention with two attention mechanisms is illustrated. The scheme behind the GATs is nothing but graph convolution of order one which learns the underlying graph from the data. This motivates us to extend the GATs to include the information from the multi-hop neighbors as in GCNNs as proposed in [16].

Remark. The number of learnable parameters to compute the attention coefficients per layer in GAT is at most F^2+2F where $F = \max_l F_l$, as there needs $F_l \times F_{l-1}$ parameters in $\mathbf{H}$ and $2F_l$ parameters in $\mathbf{a}$. Computationally, the GAT is efficient: the computation of the attention coefficients can be parallelized across all edges, as well as the computation of the output features. The computation complexity of an L-layer GAT to compute the output features per layer (7.33) can be expressed as $\mathcal{O}(F(NF+M))$ since Φ respects the sparsity of the graph. Furthermore, as the attention coefficients α_{ij} and α_{ji} for node i and j are not necessarily equal to each other, the permutation equivariance does not hold for GATs in general.

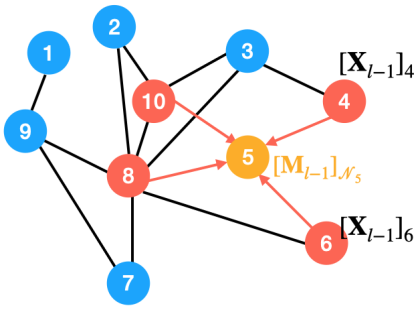
Graph Convolutional Attention Networks (GCATs). To account for the information from the multi-hop neighbors in GATs, we can directly build the convolutional attention network, specified as follows at layer l

$$\mathbf{X}_l = \sigma\Big[\sum_{k=0}^{K}\Phi^k\mathbf{X}_{l-1}\mathbf{H}_k\Big] \tag{7.37}$$

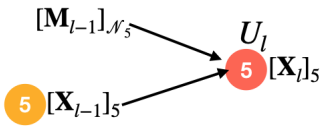
where we utilize the graph filter convolution idea, and $\Phi^k = \Phi_l^k$ and $\mathbf{H}_k = \mathbf{H}_{lk}$ are layer-dependent but we omit the layer index as we did before. Since the shifting matrix Φ shares the graph support as the shifting operator $\mathbf{S}$, (7.37) defines a GNN. In a GCAT, the matrix Φ is learned from the attention mechanism (7.31).

Remark. Note that (7.37) is a convolutional extension of the single-head attention network [cf. (7.32) and (7.33)]. The GCATs can likewise be extended with the multi-head attention mechanism proposed in [22] to improve the network capacity. In addition, the GAT framework is agnostic to the particular choice of attention mechanism.

Remark. At each layer, the computation of the attention coefficients in (7.37) depends on the $F_{l-1}\times F_l$ parameters in $\mathbf{H}_k$ and the $2F_l$ parameters in attention vector $\mathbf{a}$. This leads to the number of learnable parameters at most $F^2+2F+F^2(K+1)$, which depends on the design choices and is independent of the number of edges. Since Φ respects the graph support, the computational complexity of implementing (7.37) and its parameterization is of order $\mathcal{O}(F(NF+KM))$. As GCAT is a generalized version of GAT, the attention mechanism leads to that permutation equivariance does not hold for GCAT.



(a) Node 5 (in yellow color) aggregates information from its neighbours (in red color) $\mathcal{N}_5 = \{4,6,8,10\}$ to form the message $[\mathbf{M}_{l-1}]_{\mathcal{N}_5}$ through the aggregate function $A_l(\cdot)$, i.e., (7.38a).



(b) Node 5 (in yellow) update its embedding based on its previous embedding $[\mathbf{X}_{l-1}]_5$ and the message $[\mathbf{M}_{l-1}]_{\mathcal{N}_5}$ through the update function $U_l(\cdot)$, i.e., (7.38b).

Figure 7.22: Message passing illustration at node 5.

7.8.3. Message Passing Neural Networks

We have seen so far graph neural network architectures that are built from a composition of graph convolutions (graph filters) and pointwise nonlinearities as in classical convolution neural networks. But a GNN can also be motivated by using a form of *neural message passing* in which features (messages) are exchanged between nodes and their neighbors, and updated using neural networks [11].

During each message-passing iteration in a GNN, the *embedding* $[\mathbf{X}_l]_i$, the i-th row of the feature $\mathbf{X}_l$ at layer l, corresponding to the node $i \in \mathcal{V}$ is updated according to information aggregated from node i's neighborhood $\mathcal{N}_i$ based on their previous information stored in $\mathbf{X}_{l-1}$. This message-passing update at the l-th iteration can be expressed as the following two steps

$$[\mathbf{M}_{l-1}]_{\mathcal{N}_i} = A_l(\{[\mathbf{X}_{l-1}]_j, \text{for all } j \in \mathcal{N}_i\}), \text{ where } A_l(\cdot) \text{ is the aggregate function,} \tag{7.38a}$$

$$[\mathbf{X}_l]_i = U_l\big([\mathbf{X}_{l-1}]_i, [\mathbf{M}_{l-1}]_{\mathcal{N}_i}\big), \text{ where } U_l(\cdot) \text{ is the update function.} \tag{7.38b}$$

where $A_l(\cdot)$ and $U_l(\cdot)$ are arbitrary differentiable functions, i.e., neural networks, and $[\mathbf{M}_{l-1}]_{\mathcal{N}_i}$ is the "message" that is aggregated from node i's neighbors. At each iteration l of the GNN, node i aggregates the messages from its neighborhood $\mathcal{N}_i$ through the aggregate function $A_l(\cdot)$ and generates a message $[\mathbf{M}_{l-1}]_{\mathcal{N}_i}$ based on this aggregated neighborhood information. Then, node i combines the message $[\mathbf{M}_{l-1}]_{\mathcal{N}_i}$ with its previous embedding $[\mathbf{X}_{l-1}]_i$ and updates its embedding through the update function $U_l(\cdot)$ to generate the updated embedding $[\mathbf{X}_l]_i$.

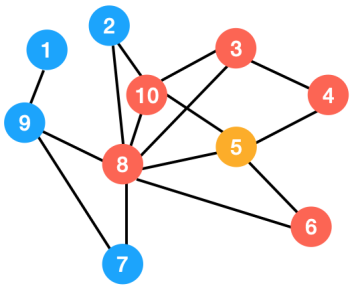
The basic intuition behind the MPNN is straightforward: at each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph. More specifically, after the first iteration, every node embedding contains information from its 1-hop neighbors; after the second iteration, every node embedding contains information from its 2-hop neighbors; and after l iterations every node embedding contains information about its l-hop neighbors.

Example. Here we illustrate the message passing neural network at node 5 in the graph in Figure 7.22. Suppose the current state at node 5 is $[\mathbf{X}_{l-1}]_5$, in the first step of the message passing (7.38a), node 5 collect the node embeddings from its direct neighbours $\mathcal{N}_5 = \{4,6,8,10\}$, then node 5 aggregates them to form the message $[\mathbf{M}_{l-1}]_{\mathcal{N}_5}$, as shown in Figure 7.22a. In the second step (7.38b), node 5 updates its embedding based on its previous state $[\mathbf{X}_{l-1}]_5$ and the message $[\mathbf{M}_{l-1}]_{\mathcal{N}_5}$ collected from its neighbours, as shown in Figure 7.22b. By iterating such message passing, each node will contain the information from its further neighbours. For example, in Figure 7.23, node 5 and node 9 will contain information of their one-hop neighbours after one message passing. Moreover, after two message passings, they will contain information of their two-hop neighbours, as in Figure 7.24.

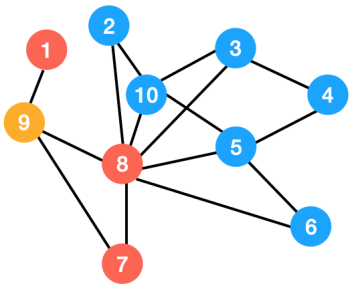
By instantiating the aggregate and update functions, we can build the most basic GNN framework, which is defined as

$$[\mathbf{X}_l]_i = \sigma\Big[\mathbf{H}_{0l}[\mathbf{X}_{l-1}]_i + \mathbf{H}_{1l}\sum_{j\in\mathcal{N}_i}[\mathbf{X}_{l-1}]_j\Big] \tag{7.39}$$

where $\mathbf{H}_{0l}, \mathbf{H}_{1l} \in \mathbb{R}^{F_l \times F_{l-1}}$ are trainable parameter matrices. Here we consider the update functions $U_l(\cdot)$ are linear transformations composed by the pointwise nonlinearities and the aggregate functions $A_l(\cdot)$ are

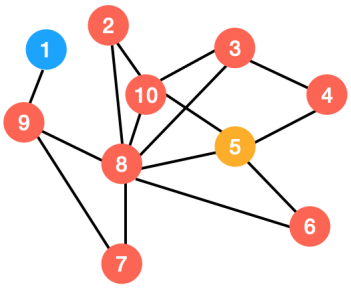


(a) Node 5 (in yellow) contains information from its one hop neighbours (in red) after one message passing.

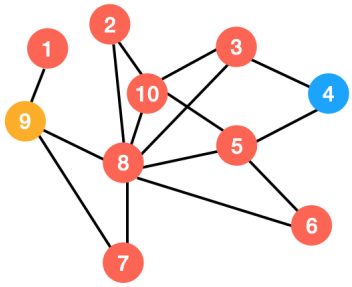


(b) Node 9 (in yellow) contains information from its one hop neighbours (in red) after one message passing.

Figure 7.23: Neighbourhood reach illustration at node 5 and 9 after one message passing in MPNN.



(a) Node 5 (in yellow) contains information from its two hop neighbours (in red) after two message passings.



(b) Node 9 (in yellow) contains information from its two hop neighbours (in red) after two message passings.

Figure 7.24: Neighbourhood reach illustration at node 5 and 9 after two message passings in MPNN.

summations. Similarly, we can write the GNN layer in a graph level as follows

$$\mathbf{X}_l = \sigma\left[\mathbf{X}_{l-1}\mathbf{H}_{0l} + \mathbf{S}\mathbf{X}_{l-1}\mathbf{H}_{1l}\right] \tag{7.40}$$

where $\mathbf{S}$ is the graph shifting matrix. From (7.40), we can see that MPNN is a special case of GCNN with $K = 1$. The main advantage of MPNN is that they are simple and enjoy also an efficient implementation. By choosing other aggregate and update functions, we can end up with the GNN forms introduced in this chapter. The basic GNN (7.40) can be derived as a neural network variation of an existing algorithm, the Weisfieler-Lehman (WL) graph isomorphism algorithm [5], which also provides us with tools to analyze the power of GNNs in a formal way through the connections to graph isomorphism testing [12].

7.9. Applications

In this section, we illustrate the applicability of GNNs in three applications, node classification, recommender systems and authorship attribution.

7.9.1. Node classification

The goal in node classification is to predict the node label, which could be a type, category, or attribute, associated with all nodes, when we are only given the true labels on a training set of nodes. For example, we want to identify if a user in a large social network with millions of users is a bot. Manually examining every user would be prohibitively expansive, so ideally we would like to have a model that could classify users as a bot or not given only a small number of manually labeled examples. It could also arise as classifying the topic of documents based on hyperlink or citation graphs based on a small number of labeled ones [12].

Here, we show the node classification problems in the CORA-ML citation network [3]. Each node represents a scientific publication, with bag-of-words feature vectors. An undirected edge is formed between two publications if one cites the other. Node labels represent the seven categories as subfields of machine learning, e.g., computational biology and natural language processing. Using the adjacency matrix to represent the graph structure can significantly improve the performance, as shown in Figure 7.25c where the GCNN performance is reported on the CORA-ML dataset. The green trend shows GCNN performance when trained with the graph

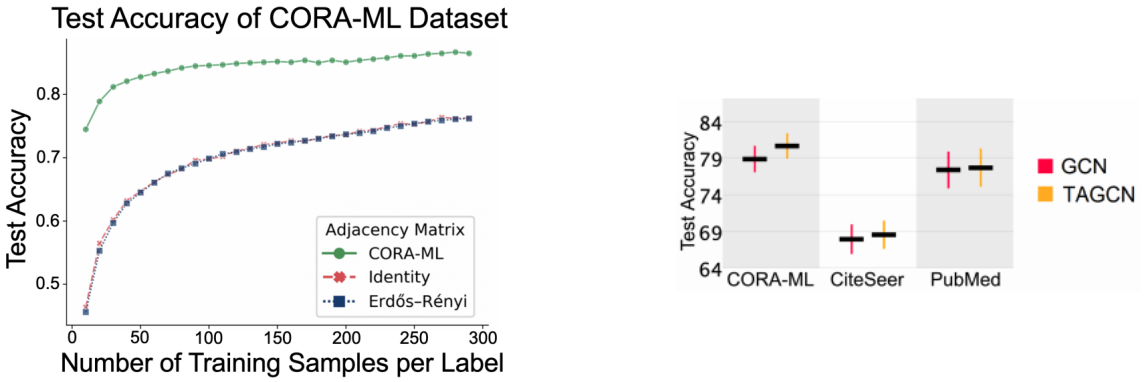


Figure 7.25: Left: Node classification accuracy on the CORA-ML citation dataset with three different graph structures, i.e., the true citation network graph, the identity matrix, and an Erdos-Renyi graph. Right: Node classification on three citation datasets with GCN and topology-adaptive GCNN (TAGCN). Both figures are from [3].

structure of CORA-ML, which significantly outperforms the same model when the graph structure is ignored (trained with an identity matrix) or generated randomly (trained with a random Erdos-Renyi graph) [3]. In Figure 7.25d, from the comparison between the GCNN with $K = 3$ (TAGCN) and the GCN (i.e., GCNN with $K = 1$), we can see that GCNN performs consistently better than GCN by considering higher-order graph filters in the graph convolution.

7.9.2. Recommender systems

We can also use graph neural networks for movie rating prediction in a recommender system. The MovieLens 100K dataset is considered [13], which contains $U = 943$ users, $I = 1,682$ movies, and 100K known ratings out of around 1,5M potential ratings. This can be seen as a rating matrix $\mathbf{X} \in \mathbb{R}^{U \times I}$ with the entry x_{ij} being the rating for the movie j given by the user i. The rating prediction task in recommender system is to predict the unknown ratings (i.e., the missing entries in the rating matrix) based on the ratings from the similar users or movies, which is essentially to complete (predict) the rating matrix if we set the missing ratings to be zeros.

From the incomplete rating matrix, we show a user-based scenario for missing rating prediction, i.e., making predictions based on the similar users. The similarity between users can be computed and stored in a similarity matrix $\mathbf{S} \in \mathbb{R}^{U \times U}$. This matrix practically constructs a similarity graph with a GSO $\mathbf{S}$. Then, the rating prediction task can be seen as a signal interpolation task on this graph $\mathbf{S}$, as we need to predict the missing values in $\mathbf{X}$. In [14], the classical collaborative filtering has been interpreted as a graph filtering problem, while graph neural networks and their variants can also perform the rating prediction task.

By considering the 200 users with the most rated movies as the nodes of a graph, the edge weights can be computed through the Pearson similarities between any two users. Each of the 1,582 movies is treated as a different graph signal whose value at a node is the rating given to that movie by a user or zero if unrated. We are interested to predict the rating of a specific user u with GNNs, i.e., to complete the u-th row of the $200 \times 1,582$ sub-rating matrix. Since this task is permutation equivariant, therefore, we expect architectures holding this property to perform better.

Different GNN architectures including GCNN, EdgeNet, node varying GNN, hybrid GNN, GAT and GCAT are considered for this task. Detailed experimental setup is shown in [16]. Table 7.1 shows the rooted mean squared error (RMSE) results for the five users with the largest number of ratings. The first thing to note is that GCAT consistently improves GAT. The latter further stresses that multi-head attentions are more needed in the GAT than in the GCAT. Second, the edge varying GNN yields the worst performance because it is not a permutation equivariant architecture. In fact, the node varying and the hybrid edge varying, which are approaches in-between permutation equivariance and local detail, work much better.

7.9.3. Authorship attribution

In the third experiment, we show the performance of the different GNN architectures in an authorship attribution problem. The task is to classify if a text excerpt belongs to a specific author or the rest authors based on word adjacency networks (WANs) [19]. A WAN is an author-specific directed graph whose nodes are function words without semantic meaning, e.g., prepositions, pronouns. The relative positioning of function words, which carries stylistic information about the author, can be used to classify the text excerpt. To capture

Table 7.1: Average RMSE on the movie graph [16].

Archit./Movie-ID	405	655	13	450	276	Average
GCNN	**1.09**	0.72	1.18	0.82	0.66	0.89
Edge var.	1.25	0.74	1.34	0.99	0.70	1.00
Node var.	1.17	**0.68**	1.19	0.84	0.67	0.91
Hybrid edge var.	**1.10**	0.72	1.27	0.80	**0.60**	0.90
GAT	1.27	0.74	1.44	0.92	0.80	1.03
GCAT	**1.09**	0.71	**1.12**	**0.77**	0.65	**0.87**

Table 7.2: Authorship attribution Test Error. The results show the average classification test error and standard deviation on 10 different training-test splits [16].

Archit./Authors	Austen	Brontë	Poe
GCNN	7.2(±2.0)%	**12.9**(±**3.5**)%	14.3(±6.4)%
Edge var.	7.1(±2.2)%	13.1(±3.9)%	**10.7**(±**4.3**)%
Node var.	7.4(±2.1)%	14.6(±4.2)%	11.7(±4.9)%
Hybrid edge var.	**6.9**(±**2.6**)%	14.0(±3.7)%	11.7(±4.8)%
GAT	10.9(±4.6)%	22.1(±7.4)%	12.6(±5.5)%
GCAT	8.2(±2.9)%	13.1 ± 3.5)%	13.6(±5.8)%

this information, a WAN can be constructed where each edge weight represents the average co-occurrence of the connected two function words. If two words are close to each other, the weight is higher than if they are further apart. The frequency count of the function words in text excerpts of 1,000 words can be seen as the signal on top of the WAN. These are the graph signals that form the dataset that is used for training and testing, as illustrated in Figure 7.26. More details on the WANs can be found in [19].

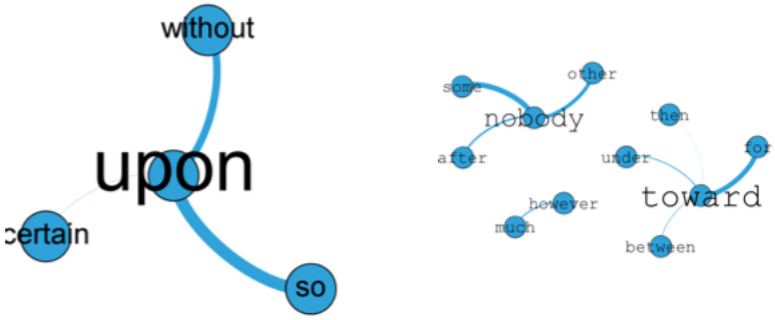


Figure 7.26: Word adjacency networks for two authors where the fonts of the words indicate the values of the signal, i.e., the frequency count, and the widths of the edges indicate the values of the edge weights. Figures are from [21].

The WANs and the word frequency count (i.e., the graph signal) serve as an author signature and allow learning the patterns of writing styles of different authors. The task is a binary classification problem where one indicates the text excerpt is written by the author of interest and zero by any other author.

Following [19], WANs for Hane Austenm Emily Brontë and Edgar Allan Poe are built. For each author, the frequency each function word pair co-appears in a window of ten words is counted. These co-appearances are imputed into an $N \times N$ matrix and normalized row-wise. The matrix is used as the shifting operator.

Following the experimental setup in [9], different GNNs are used to perform the authorship attribution problem for above three authors. Table 7.2 shows the results of this experiment. Overall, we see that the graph convolution is a solid prior to learning meaningful representations. This is particular highlighted in the improved performance of the GCAT. Furthermore, the hybrid edge varying GNN improves the accuracy of the node varying counterpart.

7.10. Summary and Further Reading

In this chapter, the basics in graph neural networks are covered. First, we discussed that graph neural networks are preferred than the fully connected neural networks due to the complexity issue. Then, we have seen

that graph filters limit the learning ability. However, graph convolutional neural networks (GCNN), as a composition of the graph filter banks and pointwise nonlinearity, are more expressive. We discussed the major components for a GCNN and showed its properties, e.g., low complexity, small number of parameters, distributed implementation and permutation equivariance.

In addition, many other graph neural networks architectures are shown to be particular cases of GCNN, for example, graph convolutional network (GCN) and simplified graph convolutional network (SGC). By considering a more expressive graph filter, we can build a more expressive GNN. EdgeNet considers edge varying graph filters which generalizes the GCNN, and its variants are discussed. Meanwhile, we also showed graph attention networks (GAT) to learn the edge weights. A different perspective of graph neural network through message passing is also discussed to help understand GCNN. In the end, three applications of GNNs are illustrated to show their abilities in node classification, recommendation and authorship attribution.

Bibliography

[1] James Atwood and Don Towsley. “Diffusion-convolutional neural networks”. In: *arXiv preprint arXiv:1511.02136* (2015).

[2] Peter W Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: *arXiv preprint arXiv:1806.01261* (2018).

[3] M. Cheung et al. “Graph signal processing and deep learning: Convolution, pooling, and topology”. In: *IEEE Signal Processing Magazine* 37 (2020), pp. 139–149.

[4] Mario Coutino, Elvin Isufi, and Geert Leus. “Advances in distributed graph filtering”. In: *IEEE Transactions on Signal Processing* 67.9 (2019), pp. 2320–2333.

[5] Brendan L Douglas. “The weisfeiler-lehman method and graph isomorphism testing”. In: *arXiv preprint arXiv:1101.5211* (2011).

[6] Federico Errica et al. “A fair comparison of graph neural networks for graph classification”. In: *arXiv preprint arXiv:1912.09893* (2019).

[7] F. Gama et al. “Convolutional Neural Network Architectures for Signals Supported on Graphs”. In: *IEEE Transactions on Signal Processing* 67.4 (2019), pp. 1034–1049.

[8] F. Gama et al. “Graphs, Convolutions, and Neural Networks: From Graph Filters to Graph Neural Networks”. In: *IEEE Signal Processing Magazine* 37.6 (2020), pp. 128–138.

[9] Fernando Gama et al. “Convolutional neural network architectures for signals supported on graphs”. In: *IEEE Transactions on Signal Processing* 67.4 (2018), pp. 1034–1049.

[10] H. Gao and S. Ji. “Graph u-nets”. In: *international conference on machine learning*. PMLR. 2019, pp. 2083–2092.

[11] Justin Gilmer et al. “Neural message passing for quantum chemistry”. In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1263–1272.

[12] William L Hamilton. “Graph representation learning”. In: *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14.3 (2020), pp. 1–159.

[13] F Maxwell Harper and Joseph A Konstan. “The movielens datasets: History and context”. In: *Acm transactions on interactive intelligent systems (tiis)* 5.4 (2015), pp. 1–19.

[14] Weiyu Huang, Antonio G Marques, and Alejandro R Ribeiro. “Rating prediction via graph signal processing”. In: *IEEE Transactions on Signal Processing* 66.19 (2018), pp. 5066–5081.

[15] B. Iancu and E. Isufi. “Towards Finite-Time Consensus with Graph Convolutional Neural Networks”. In: *2020 28th European Signal Processing Conference (EUSIPCO)*. 2021, pp. 2145–2149.

[16] Elvin Isufi, Fernando Gama, and Alejandro Ribeiro. “Edgenets: Edge varying graph neural networks”. In: *arXiv preprint arXiv:2001.07620* (2020).

[17] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).

[18] Y. Ma et al. “Graph convolutional networks with eigenpooling”. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2019, pp. 723–731.

[19] Santiago Segarra, Mark Eisen, and Alejandro Ribeiro. “Authorship attribution through function word adjacency networks”. In: *IEEE Transactions on Signal Processing* 63.20 (2015), pp. 5464–5478.

[20] Santiago Segarra, Antonio G Marques, and Alejandro Ribeiro. “Optimal graph-filter design and applications to distributed linear network operators”. In: *IEEE Transactions on Signal Processing* 65.15 (2017), pp. 4117–4131.

[21] Tomas SIPKO. “Identifying author fingerprints in texts via graph neural networks”. Master’s Thesis. Delft University of Technology, 2020.

[22] Petar Veličković et al. “Graph attention networks”. In: *arXiv preprint arXiv:1710.10903* (2017).

[23] Felix Wu et al. “Simplifying graph convolutional networks”. In: *International conference on machine learning*. PMLR. 2019, pp. 6861–6871.

[24] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *IEEE transactions on neural networks and learning systems* (2020).

[25] Keyulu Xu et al. “How powerful are graph neural networks?” In: *arXiv preprint arXiv:1810.00826* (2018).

[26] R. Ying et al. “Hierarchical graph representation learning with differentiable pooling”. In: *arXiv preprint arXiv:1806.08804* (2018).

8

Graph-Time Representations

Data are often time-dependent as they arise from domains having a time-varying component. This calls for techniques to deal with time-series: observational data values obtained at different time instants. When multiple time-series are simultaneously observed, we talk about multi-variate time series. If the time series are the result of a process happening over a physical network, or the relationship between the different time series can be captured by a graph, we can use such mathematical tool to represent their structure. In this view, each time series is seen as a signal evolving over the nodes of a graph, and the overall picture enables us to define the concept of *time-varying graph signal* or *graph-time signal*. See Figure 8.1 for an illustrative example.

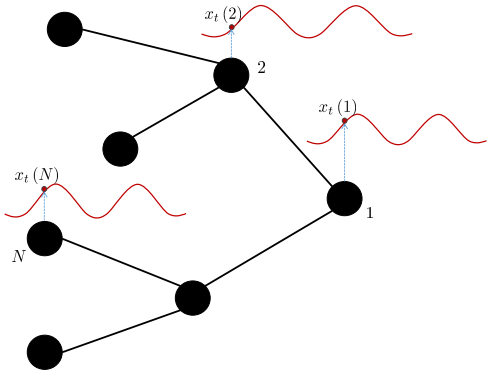


Figure 8.1: A time-varying graph signal. Each node of the graph has an associated time-series. For a fixed time instant t, we have on top of the graph the usual definition of graph signal. Informally, we may define a time-varying graph signal as a graph signal indexed by the time.

In a temperature sensor network, the nodes of the graph represent measurement stations and the edges represent the relationships between the different stations (for instance the physical pairwise distance). This is an example where the data are the temperatures over time and over space in the network. We expect adjacent geographical locations to share similar behavioral patterns, either in space and/or in time. Exploring this spatio-temporal coupling can help processing the time series by accounting for a sparse underlying topology given by the graph. Applications where the latter plays a role include:

- *Forecasting*: the goal is to forecast the future values of the time process from its past history (time dimension intrinsic of the data) and the knowledge of the structure of the graph (space dimension is a prior of the data) [6, 9].
- *Interpolation*: the goal is to reconstruct the signal missing values from a subset of measurements Interpolation can happen either in time and in space [11].
- *Link prediction*: the goal is to predict whether two nodes of the graph share a link, i.e. they are in a relationship, under a specified criterion. This extends to the more general framework of topology identification, in which the goal is to learn the entire graph structure [1].

- *Anomaly detection*: the goal is to detect unexpected behaviors/pattern of the data, i.e. events that deviate from the regular behaviors of the data/process. These irregular events can be local both spatially and temporally.

There are two main approaches to handle time-varying data over graphs, which we briefly introduce in the sequel and elaborate more in the detail in the upcoming sections.

- *Graph-based temporal recursions:* in this model, the data value at a specific node and at a certain time instant depends on its own history and on the history of its neighbors. These models can be used to forecast signal values exploiting the graph geometry. The main benefits of this approach is the reduction of the degrees of freedom with respect to the graph-free models, yielding to a lower complexity and a number of parameters independent of the size of the graph [6].

- *Product graphs*: in this model, each time instant in the time series is considered as a scalar value over a graph. Then the different graphs are connected with each other based on the temporal contiguity [13]. By building a single graph representing spatio-temporal instances, we can translate to this domain the processing tools uses for processing signals over static graphs. Likewise, they can be used to decompose or approximate a large graph into smaller (factor) graphs [7]. The benefits of this approach are the theoretical properties of product graphs, relying on a solid mathematical background, enabling parallel and vectorized operations on the smaller graphs parameters, and its high interpretability. On the downside, product graphs are large and may sometimes be computationally more difficult to handle.

8.1. Graph-based Temporal Recursions

Time-varying graph data models have to take into account the underlying network structure. This because general multivariate data can be modeled through vector autoregressive (VAR) recursions [8], but these models do not capture any structure present in the data. The challenge is then to incorporate such graph structure into the model to make the latter more effective.

8.1.1. Vector Autoregressive Model

For a scalar time-varying signal x_t, an autoregressive (AR) model of order P, denoted as AR(P), is defined as:

$$x_t = \sum_{p=1}^{P} h_p x_{t-p} + \varepsilon_t \tag{8.1}$$

where $h_1, \dots, h_P$ are the scalar parameters and ε_t is white noise. Model (8.1) is a linear model and expresses the signal value x_t as the linear combination of its previous values $x_{t-P:t-1} := \{x_{t-P}, \dots, x_{t-1}\}$ up to a finite memory P. The noise term ϵ_t includes the part of x_t that the model cannot represent. This model has been successfully applied to forecasting.

Consider now a multivariate time series $\mathbf{x}_t \in \mathbb{R}^N$. A vector autoregressive (VAR) model of order P, denoted as VAR(P), is defined as:

$$\mathbf{x}_t = \sum_{p=1}^{P} \mathbf{H}_p \mathbf{x}_{t-p} + \boldsymbol{\varepsilon}_t \tag{8.2}$$

where $\mathbf{x}_t \in \mathbb{R}^N$ is the multivariate data signal at time t, $\mathcal{H} := \{\mathbf{H}_1, \dots, \mathbf{H}_P\}$ are the $N \times N$ parameter matrices of the model and $\boldsymbol{\varepsilon}_t \in \mathbb{R}^N$ is a white noise vector. The model expresses the current value $\mathbf{x}_t$ as the linear combination of the P past realizations $\mathbf{x}_{t-P:t-1} := \{\mathbf{x}_{t-P}, \dots, \mathbf{x}_{t-1}\}$, thus generalizing the single variable AR process in (8.1)[1]. Each one of the $N \times N$ matrices $\mathbf{H}_p$ contains N^2 parameters, expressing the direct influence of the p-th past realization into the current one.

Parameter Estimation. We can estimate the P matrices $\mathbf{H}_p$ in an empirical risk minimization fashion. We use the map $\phi(\mathbf{x}_{t-P:t-1}; \mathcal{H}) = \sum_{p=1}^{P} \mathbf{H}_p \mathbf{x}_{t-p}$ belonging to the class of linear functions, a loss function $\ell(\mathbf{x}_t, \hat{\mathbf{x}}_t)$ and a training set $\mathcal{T} = [\mathbf{x}_1, \dots, \mathbf{x}_{|\mathcal{T}|}]$ containing the graph signals of interest. Then, the model parameters can be obtained by solving the optimization problem:

$$\underset{\mathcal{H}}{\operatorname{argmin}} \frac{1}{|\mathcal{T}| - P} \sum_{t=P+1}^{|\mathcal{T}|} \ell(\mathbf{x}_t, \phi(\mathbf{x}_{t-P:t-1}; \mathcal{H})) + \lambda g(\mathcal{H}) \tag{8.3}$$

[1] From now on, we will directly focus on the multivariate case, which is our situation of interest.

where $g(\cdot)$ is a possible regularization term, e.g., imposing sparsity or smoothness in the coefficients $\mathscr{H}$, and $\lambda \geq 0$. Solving problem (8.3) is tantamount to the estimation of PN^2 parameters. This is practical when the number of time series N is contained but it becomes a challenge for a large N. To overcome this challenge, parametric models for the matrices $\mathbf{H}_p$ are necessary, and this is where graphs come into play.

8.1.2. Graph-Vector Autoregressive Model

When the time series of interest reside over a graph, we can exploit this structure to reduce the PN^2 parameters of (8.3). In particular, let graph $\mathscr{G} = \{\mathcal{V}, \mathcal{E}, \mathbf{S}\}$ model the network, where $\mathcal{V}, \mathcal{E}, \mathbf{S}$ are the node set, edge set, and GSO, respectively. Let also $\mathbf{x}_t$ be a multivariate time varying graph signal in which ith entry $x_t(i)$ is the signal value at node i. The graph autoregressive (G-VAR) model for $\mathbf{x}_t$ reads as:

$$\mathbf{x}_t = -\sum_{p=1}^{P} \mathbf{H}_p(\mathbf{S})\mathbf{x}_{t-p} + \boldsymbol{\varepsilon}_t = -\sum_{p=1}^{P}\sum_{k=0}^{K} h_{kp}\mathbf{S}^k\mathbf{x}_{t-p} + \boldsymbol{\varepsilon}_t. \tag{8.4}$$

In other words,

$$\mathbf{x}_t = -\left(\sum_{k=0}^{K} h_{k1}\mathbf{S}^k\mathbf{x}_{t-1} + \sum_{k=0}^{K} h_{k2}\mathbf{S}^k\mathbf{x}_{t-2} + \ldots + \sum_{k=0}^{K} h_{kP}\mathbf{S}^k\mathbf{x}_{t-P}\right) + \boldsymbol{\varepsilon}_t \tag{8.5}$$

The resulting graph signal $\mathbf{x}_t$ is thus expressed as the linear combination of the P previous graph signals $\mathbf{x}_{t-P}, \ldots, \mathbf{x}_{t-1}$, each one processed with a different graph filter. Comparing (8.4) with (8.2), we notice how the P unstructured VAR matrices $\mathbf{H}_p$ embed now P different structured graph filters $\mathbf{H}_p(\mathbf{S})$. Assuming each graph filter to be of order K, the total number of parameters for model (8.4) is PK, which is much smaller than the PN^2 parameters for the VAR model in (8.2). Moreover, the signal value $x_t(i)$ at node i depends now on the process values with a temporal window of P and a graph window of K, i.e., $x_t(i)$ depends on the past P values of the nodes that are within K hops from node i. Increasing K enables to capture more spatial details of the graph signal, while increasing P enables to extend the memory of the filter in time. This notion of *coverage* in time and space is illustrated in Figure 8.2, for a G-VAR model with $P = 2$ and $K = 1$.

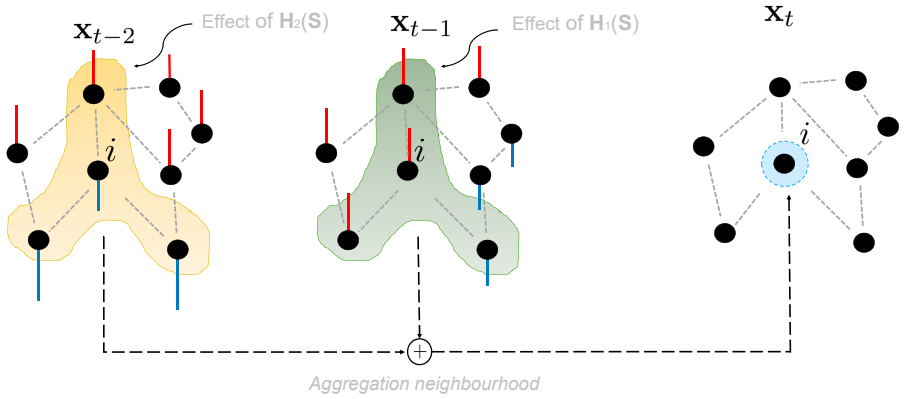


Figure 8.2: The signal value at node i at time instant t depends from itself and its 1-hop neighbors up to $P = 2$ previous time instants.

Joint Causality Graph-Time. The former G-VAR strategy considers the graph filters $\mathbf{H}_p(\mathbf{S})$ as a mathematical object to scale down the parameters and complexity of the VAR model (8.2). However, it does not impose any link between the temporal order memory P and the spatial one K. Contrarily, considering $P < K$ may allow interpreting the G-VAR recursion (8.4) in the context of a joint graph-time causality. Efforts in this direction has been put forth in [9], by considering the recursion:

$$\mathbf{x}_t = \sum_{p=1}^{P} \mathbf{H}_p(\mathbf{S})\mathbf{x}_{t-p} + \boldsymbol{\varepsilon}_t = \sum_{p=1}^{P}\sum_{k=0}^{p} h_{kp}\mathbf{S}^k\mathbf{x}_{t-p} + \boldsymbol{\varepsilon}_t, \tag{8.6}$$

Equation (8.6) can be seen as a specialization of the model (8.4) constraining the order of each filter $\mathbf{H}_p(\mathbf{S})$ to have a degree of p, i.e. dependent of the lag considered.

ERM Forecasting. Likewise problem (8.3), we can estimate the parameters of the G-VAR in an ERM fashion. Respectively, let $\mathbf{H}_{KP} := [\mathbf{h}_1, \dots, \mathbf{h}_P] \in \mathbb{R}^{K+1 \times P}$ be the matrix containing in each column p the parameters relative to the pth filter $\mathbf{H}_p(\mathbf{S})$, i.e. the $K+1$ filter coefficients $\mathbf{h}_p = \{h_{0p}, \dots, h_{Kp}\}$. We can then solve the equivalent of problem (8.3) with respect to $\mathbf{H}_{KP}$, which has the form:

$$\mathbf{H}^{\star}_{KP} = \underset{\mathbf{H}_{KP}}{\operatorname{argmin}} \frac{1}{|\mathcal{T}| - P} \sum_{t=P+1}^{|\mathcal{T}|} \ell(\mathbf{x}_t, \phi(\mathbf{x}_{t-1:t-P}; \mathbf{H}_{KP})) + \lambda g(\mathbf{H}_{KP}) \tag{8.7}$$

where $g(\cdot)$ is a possible regularization term and $\lambda \geq 0$. Problems (8.3) and (8.7) are very similar in terms of structure and formulation. However, problem (8.7) is a model that has far fewer parameters with respect to (8.3), which can be accurately estimated with a training set of a smaller cardinality.

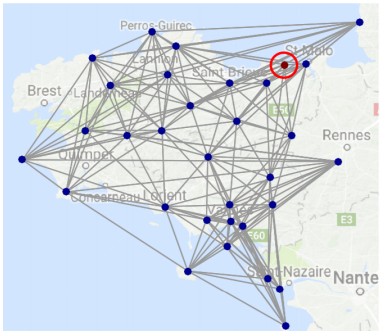
8.1.3. Applications

Here we describe two dataset where the approaches shown above have successfully been used to forecasting. We redirect the interested reader to [6] for the numerical results and a thorough explanation of the two dataset.

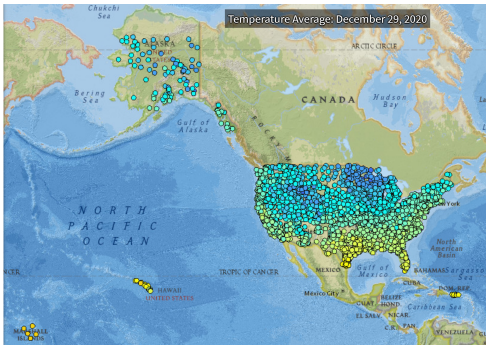
Molene Dataset. The Molene data set contains hourly temperature measurements of $N = 32$ weather stations near Brest (France) for $T = 744$ hours. We consider a geometric graph (illustrated in Fig. 8.3a) built from the node coordinates using the default nearest neighbor approach. This graph follows a 10-NN construction and the weight of the edge (i, j) follows the Gaussian kernel:

$$[\mathbf{W}]_{ij} = \exp(-\operatorname{dist}(i, j)/\overline{\operatorname{dist}}) \tag{8.8}$$

where $\mathbf{W}$ is the weighted adjacency matrix, $\operatorname{dist}(i, j)$ is the Euclidean distance between stations i and j and $\overline{\operatorname{dist}}$ is the average Euclidean distance of all sensors. The Laplacian matrix, obtained from $\mathbf{W}$, is normalized to have a unitary spectral norm.



(a) Molene graph.



(b) NOAA dataset. Each point in the map represents the location of the weather station. The color represent the intensity of the measured quantity, in this case the average daily temperature on December 29th, 2020.

NOAA Dataset. This is another temperature data set and comprises of hourly temperature recordings at $N = 109$ stations across the United States in 2010 (illustrated in Fig. 8.3b) for a total of $T = 8759$ hours. Similarly to the Molene dataset, the graph structure relies on the 7-NN geographical distances. The combinatorial Laplacian is used to represent the graph connectivity and the in-sample mean is subtracted from the raw data before preprocessing.

8.2. Product Graph Representation

Methods in the above section leverage temporal recursions of time series and embed into them graph filters. Differently, product graphs aim at representing a multivariate temporal recursion as a static graph signal over a larger graphs capturing both spatial and temporal relations [3]. Product graphs arise in several fields, including biology and finance. They can decompose large graphs, alleviating storage requirements and computational efforts.

A scalar time series can be seen as a graph signal residing on top of a line graph, where each node of the graph represents the discrete time index. When we have a multivariate time series residing on top of a network, two graphs can be used to represent such setting: *i)* the line graph for each time series; and *ii)* the spatial

graph, to capture dependencies among time series. Product graphs enable us to have a unified and combined picture of this spatio-temporal dataset, by combining the two graphs into a larger one. See Figure 8.4 for an illustration.

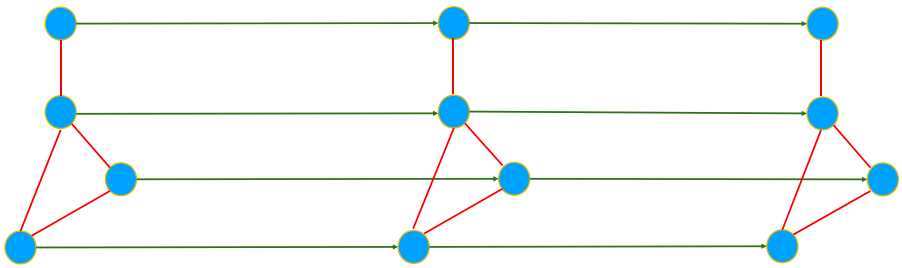


Figure 8.4: An example of product graph between a spatial graph, possibly representing a sensor network, and a line graph, representing the time series. The Cartesian product graph between the spatial graph and the line graph leads to the larger graph, which is a combination of the two. Different types of product graph lead to different combinations of edges.

8.2.1. Types of Product Graphs and Graph Signals

Consider an $N \times 1$ multivariate signal $\mathbf{x}_t$ collected over T time instances in matrix $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_T]$, such as sensor recordings in a sensor network. Signals in X have spatiotemporal relations, which if fully-exploited serve as a powerful inductive bias to learn representations [26]. When signal $\mathbf{x}_t$ has an (hidden) underlying structure, we can represent its *spatial* relations through a spatial graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of N nodes in set $\mathcal{V} = \{1, \ldots, N\}$ and $|\mathcal{E}|$ edges in set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Signal $\mathbf{x}_t = [x_t(1), ..., x_t(N)]\top$ are a collection of values $x_t(i)$ residing on node i at time t. Likewise, we can capture the *temporal* relations by viewing each row $\mathbf{x}^i = [x_1(i), \ldots, x_T(i)]^\top$ of $\mathbf{X}$ as a graph signal over the nodes of a temporal graph $\mathcal{G}_T = (\mathcal{V}_T, \mathcal{E}_T)$ of T nodes $\mathcal{V}_T = \{1, \ldots, T\}$ and $|\mathcal{E}_T|$ edges $\mathcal{E}_T = (t, t')$. Set $\mathcal{E}_T$ contains an edge if signals at time instances t and t' are related. Examples for $\mathcal{G}_T$ are the directed line graph that assumes signal $\mathbf{x}_t$ depends only on the former instance $\mathbf{x}_{t-1}$, the cyclic graph that accounts for periodicity, or any other graph that encodes the temporal dependencies in $\mathbf{X}$. We represent graphs $\mathcal{G}$ and $\mathcal{G}_T$ through their respective graph shift operator matrices $\mathbf{S} \in \mathbb{R}^{N \times N}$ and $\mathbf{S}_T \in \mathbb{R}^{T \times T}$; e.g., adjacency [12].

Given graphs $\mathcal{G}$ and $\mathcal{G}_T$, we can capture the spatio-temporal relations in $\mathbf{X}$ through the product graph:

$$\mathcal{G}_\diamond = \mathcal{G}_\mathcal{T} \diamond \mathcal{G} = (\mathcal{V}_\diamond, \mathcal{E}_\diamond, \mathbf{S}_\diamond), \tag{8.9}$$

where the node set $\mathcal{V}_\diamond = \mathcal{V}_T \times \mathcal{V}$ is the Cartesian product between $\mathcal{V}_T$ and $\mathcal{V}$ (hence of cardinality $|\mathcal{V}_\diamond| = NT$), and where the edge set $\mathcal{E}_\diamond \subseteq \mathcal{V}_\diamond \times \mathcal{V}_\diamond$ and the $NT \times NT$ graph shift operator $\mathbf{S}_\diamond$ are dictated by the type of product graph. In particular, popular product graphs are the Kronecker, Cartesian, and strong product [13, 2], defined next.

- *Kronecker product*: the graph shift operator is $\mathbf{S}_\otimes = \mathbf{S}_T \otimes \mathbf{S}$. The edge set cardinality for this graph is $|\mathcal{E}_\otimes| = |\mathcal{E}||\mathcal{E}_T|$
- *Cartesian product*: the graph shift operator is $\mathbf{S}_\times = \mathbf{S}_T \otimes \mathbf{I}_N + \mathbf{I}_T \otimes \mathbf{S}$ with $\mathbf{I}_T$ ($\mathbf{I}_N$) the $T \times T$ ($N \times N$) identity matrix. The edge set cardinality for this graph is $|\mathcal{E}_\times| = T|\mathcal{E}| + N|\mathcal{E}_T|$.
- *Strong product*: the graph shift operator is $\mathbf{S}_\boxtimes = \mathbf{S}_T \otimes \mathbf{I}_N + \mathbf{I}_T \otimes \mathbf{S} + \mathbf{S}_T \otimes \mathbf{S}$. The edge set cardinality for this graph is $|\mathcal{E}_\boxtimes| = |\mathcal{E}||\mathcal{E}_T| + T|\mathcal{E}| + N|\mathcal{E}_T|$

In Figure 8.5, we show the three product graphs applied to two different lines graph, composed respectively of $N = 3$ and $N = 4$ nodes.

Parametric Product Graph. All the aforementioned product graphs can be generalized through a parametric model, called parametric product graph [10], with shift operator:

$$\mathbf{S}_\diamond = \sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} \left(\mathbf{S}_T^i \otimes \mathbf{S}^j \right), \tag{8.10}$$

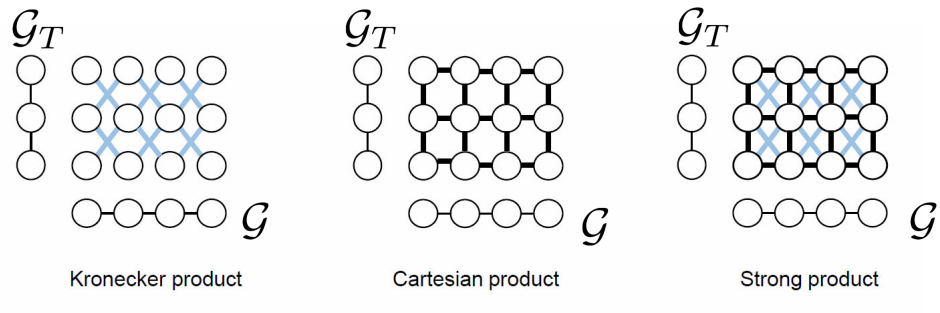


Figure 8.5: Cartesian, strong and product graphs applied to two lines graph of $N = 3$ and $N = 4$ nodes, respectively.

where $\{s_{ij}\}$ are scalars that shape the formation of the edges, and "$\otimes$" is the Kronecker product. The parametric product graph captures the spatio-temporal coupling with the four scalars s_{ij} . If all s_ijs are non-zero, the parametric product graph has $|\mathcal{E}_\diamond| = |\mathcal{E}_\boxtimes| + NT$ edges, which are NT more edges than the strong product because of self-loops ($s_{00} \neq 0$). Column-vectorizing $\mathbf{X}$ yields a product graph signal $\mathbf{x}_\diamond = \text{vec}(\mathbf{X}) \in \mathbb{R}^{NT}$ in which node $i_\diamond \in \mathcal{V}_\diamond$ represents the space-time location (i,t) with value $x_t(i)$, i.e., the $i_\diamond$th entry of $\mathbf{x}_\diamond$. Our goal is to exploit the coupling product graph signal–product graph to learn spatiotemporal representations in a form akin to temporal or graph convolutional neural networks.

8.2.2. Convolutional Filter on Product Graphs

Given the spatio-temporal product graph $\mathcal{G}_\diamond = \mathcal{G}_T \diamond \mathcal{G} = (\mathcal{V}_\diamond, \mathcal{E}_\diamond, \mathbf{S}_\diamond)$ and the respective product signal $\mathbf{x}_\diamond$, we can now perform convolutional filtering over the product graph in a form akin to the convolutional filters, introduced in Chapter **??**. Specifically, the output of a convolutional filter of order K over the parametric product graph $\mathbf{S}_\diamond$ is:

$$\mathbf{y}_\diamond = \sum_{k=0}^{K} h_k \mathbf{S}_\diamond^k \mathbf{x}_\diamond = \sum_{k=0}^{K} h_k \left(\sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} \left(\mathbf{S}_T^i \otimes \mathbf{S}^j \right) \right)^k \mathbf{x}_\diamond \tag{8.11}$$

where $h_0, \ldots, h_K$ are the filter parameters. Like standard graph filtering we are now again performing shift-and-sum operations but now over a product graph that captures spatio-temporal representations.

If parameters s_ij are fixed (i.e., the product graph), we can work directly with $\mathbf{S}_\diamond$, which has a sparsity of order $|\mathcal{E}_\diamond| = NT + N|\mathcal{E}_T| + T|\mathcal{E}| + |\mathcal{E}_T||\mathcal{E}|$. Computing output $\mathbf{y}_\diamond$ can be achieved by using the well-known recursive implementation of shifting signals over a graph [5], subject to the linear cost $\mathcal{O}(K|\mathcal{E}_\diamond|)$.

Spatio-temporal neighborhood. Operation (8.11) respects the scale-shift-and-sum principle; in essence it is a graph filter. However its behavior depends now on the filter coefficients $\{h_k\}$ and the product graph parameters $\{s_{ij}\}$. That is, the notion of *neighborhood* depends now on the type of product graph adopted (other than the order of the filter). In particular, consider the graph $\mathcal{G}$ of $N = 4$ nodes and a temporal line graph $\mathcal{G}_T$ of $T = 3$ nodes, illustrated in Figure 8.6.

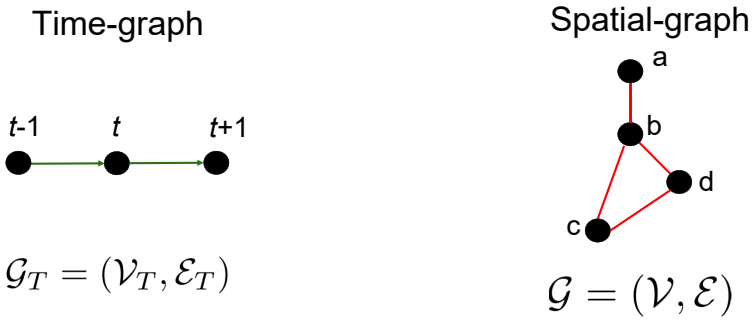


Figure 8.6: Graph $\mathcal{G}_T$ of $T = 3$ nodes and graph $\mathcal{G}$ of $N = 4$ nodes. Different choices of product graph lead to a different set of edges for the graph $\mathcal{G}_\diamond$ of NT nodes.

- *Cartesian.* For the Cartesian product, the resulting graph is shown in Figure 8.7. To illustrate the notion

of neighborhood, consider node $(t+1,d)$ in the orange circle of the figure. If the order of the filter is $K=2$, the filter recursion aggregates at node $(t+1,d)$ the values of its 2-hop neighbors (7 nodes in total, enclosed in the yellow circles) and of itself. If the order of the filter is $K=3$, the filter recursion aggregates at node $(t+1,d)$ the values of its 2-hop neighbors (yellow circles), of its 3-hop neighbors (green circles) and of itself. Notice how, for $K=2$, only the node $(t-1,d)$ among the nodes $\{(t-1,j)\}_{j=a}^{d}$ contributes to the value of $(t+1,d)$.

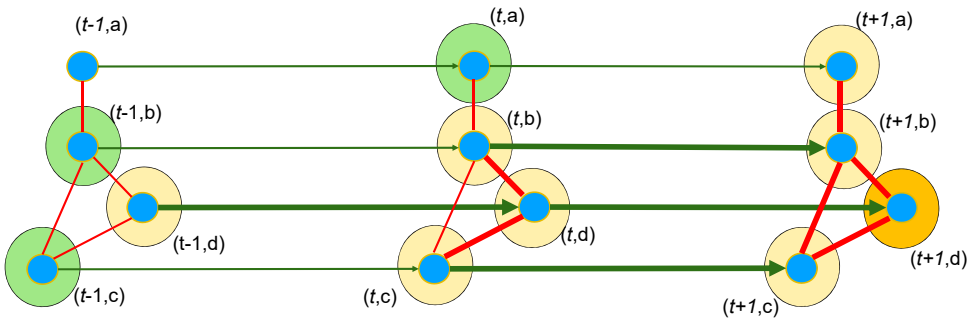


Figure 8.7: Cartesian product graph. Consider node $(t+1,d)$ in the orange circle of the figure. If the order of the filter is $K=2$, the filter recursion aggregates at node $(t+1,d)$ the values of its 2-hop neighbors (7 nodes in total, enclosed in the yellow circles) and of itself. If the order of the filter is $K=3$, the filter recursion aggregates at node $(t+1,d)$ the values of its 2-hop neighbors (yellow circles), of its 3-hop neighbors (green circles) and of itself.

- *Strong.* For the strong product, the resulting graph is shown in Figure 8.8. Here, considering again node $(t+1,d)$, the recursion with a filter order of $K=2$, aggregates at node $(t+1,d)$ the values of all the other nodes and of itself. Thus, differently from the Cartesian product, the strong product has a larger neighborhood.

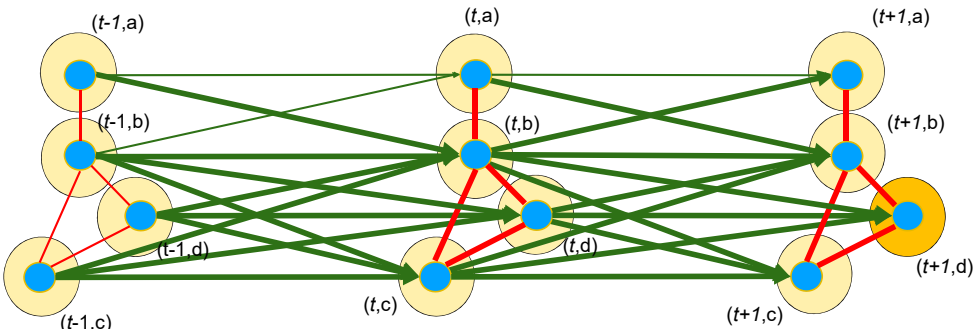


Figure 8.8: Strong product. The recursion with a filter order of $K=2$, aggregates at node $(t+1,d)$ the values of all the other nodes and of itself. Thus, differently from the Cartesian product, the strong product has a larger neighborhood.

It is clear how the choice of a product graph to use is a difficult one, since it exerts a high influence on the filtering result. To deal with this *model selection* problem, one may consider the parametric product graph (8.10) instead, and learn the parameters $\{s_{ij}\}$ with, e.g., cross-validation.

Filter Frequency Response. We can characterize the filter frequency response with respect to the product graph decomposition. Here, for simplicity, we consider the Cartesian product graph and exploit the eigendecomposition of the shift operator $\mathbf{S}=\mathbf{U}\mathbf{\Lambda}\mathbf{U}^{-1}$ and of $\mathbf{S}_T=\mathbf{U}_T\mathbf{\Lambda}_T\mathbf{U}_T^*$. Recall that $\mathbf{U}_T^*$ is the complex conjugate of the discrete Fourier transform (DFT) matrix and $\mathbf{\Lambda}_T$ is the diagonal matrix containing the eigenvalues $[\mathbf{\Lambda}_T]_{tt}=e^{-j2\pi(t-1)/T}$ with $j=\sqrt{-1}$ and $t=1,\dots,T$.

Because the (Cartesian) product-graph shift operator $\mathbf{S}_\times=\mathbf{S}_T\otimes\mathbf{I}_N+\mathbf{I}_T\otimes\mathbf{S}$ is the linear combination of

Kronecker product(s), we can eigendecompose it by exploiting the Kronecker properties (see [2]):

$$\begin{aligned}\mathbf{S}_{\times} &= \mathbf{S}_T \otimes \mathbf{I}_N + \mathbf{I}_T \otimes \mathbf{S} \\ &= \left(\mathbf{U}_T \boldsymbol{\Lambda}_T \mathbf{U}_T^*\right) \otimes \mathbf{I}_N + \mathbf{I}_T \otimes \left(\mathbf{U} \boldsymbol{\Lambda} \mathbf{U}^*\right) \\ &= \underbrace{(\mathbf{U}_T \otimes \mathbf{U})}_{\mathbf{U}_\diamond} \underbrace{(\boldsymbol{\Lambda}_T \times \boldsymbol{\Lambda})}_{\boldsymbol{\Lambda}_\diamond} \underbrace{(\mathbf{U}_T \otimes \mathbf{U})^*}_{\mathbf{U}_\diamond^*}\end{aligned} \tag{8.12}$$

where $\mathbf{U}_\diamond = \mathbf{U}_T \otimes \mathbf{U}$ is the (i)GFT of the Cartesian product graph shift operator and it is also known as joint time-vertex Fourier transform (JFT).

Because $\mathbf{U}_\diamond$ is a unitary matrix, operations such as filtering can be performed in the (product graph) frequency domain, similar to what we did in previous chapters. In particular, the joint filtered version $\mathbf{y}_\diamond$ of $\mathbf{x}_\diamond$ with the filter $h(\mathbf{S}_\diamond)$ corresponds again to an element-wise multiplication, this time in the joint frequency representation. That is,:

$$\mathbf{y}_\diamond = h(\mathbf{S}_\diamond)\,\mathbf{x}_\diamond \triangleq \mathbf{U}_\diamond h(\boldsymbol{\Lambda}, \boldsymbol{\Lambda}_T)\,\mathbf{U}_\diamond^{\mathrm{H}} \mathbf{x}_\diamond \tag{8.13}$$

where $h(\boldsymbol{\Lambda}_T, \boldsymbol{\Lambda})$ is the joint time-vertex frequency response of the filter, with entries $[h(\boldsymbol{\Lambda}_T, \boldsymbol{\Lambda})]_{kk} = h\left(e^{-j2\pi(t-1)/T}, \lambda_i\right)$ and $k = N(t-1)+i$ for $i = 1,\ldots,N$ and $t = 1,\ldots,T$.

We can see how the product-graph representation is a natural extension of the graph signal processing framework and, as such, the tools described in earlier chapters can be applied here as well.

Tikhonov Regularization over Product Graphs. As discussed in the earlier chapters, regularization techniques are necessary to enforce a prior about the data for tasks such as denoising or interpolation. Consider a signal denoising problem, e.g. sensor networks , in which the signal observation $\mathbf{y}_\diamond$ is corrupted by ambient noise and we want to recover the signal $\mathbf{x}_\diamond$ given the prior assumption that $\mathbf{x}_\diamond$ is smooth with respect either the time and the graph dimensions. This task can be addressed by means of Tikhonov regularization, where the estimate $\mathbf{x}_\diamond$ is the solution of the problem:

$$\mathbf{x}_\diamond^\star = \underset{\mathbf{x}_\diamond \in \mathbb{R}^{NT}}{\operatorname{argmin}} \|\mathbf{y}_\diamond - \mathbf{x}_\diamond\|_2^2 + \gamma \mathbf{x}_\diamond^\top \mathbf{L}_\diamond \mathbf{x}_\diamond \tag{8.14}$$

for some scalar $\gamma \geq 0$. The cost function is a trade-off between a data fidelity term $\|\cdot\|_2^2$ and the regularization term $\mathbf{x}_\diamond^\top{}_\diamond \mathbf{x}_\diamond$, which ensures $\mathbf{x}_\diamond$ to be smooth over the product graph $\mathbf{S}_\diamond$. Likewise the Tikhonov problem in Chapter .5, the solution of such problem is:

$$\mathbf{x}_\diamond = (\mathbf{I}_{NT} + \gamma \mathbf{L}_\diamond)^{-1} \mathbf{y}_\diamond \tag{8.15}$$

In other words, the optimal solution $\mathbf{x}_\diamond$ is a graph filtering of $\mathbf{y}_\diamond$ with the graph filter $\mathbf{H} = (\mathbf{I}_{NT} + \gamma \mathbf{L}_\diamond)^{-1}$.

8.2.3. Advantages and Challenges of Product Graph Representation

Advantages

- Product graphs allow for a solid mathematical background and give a good intuition for the representation (i.e. a larger graph). For instance, as it follows from (8.10) they allow us to define a convolution operation over both the graph and temporal domain in a principled way.
- Product graphs allow us to define a graph-time Fourier transform:

$$\hat{\mathbf{x}}_\diamond = \mathbf{U}_\diamond^{-1} \mathbf{x}_\diamond = (\mathbf{U}_T \otimes \mathbf{U})^{-1} \mathbf{x}_\diamond = \mathrm{vec}(\mathbf{U}^{-1} \mathbf{X} \mathbf{U}_T^*) \tag{8.16}$$

  in which we clearly see how the transform operates on the columns $\{\mathbf{x}_i\}_{i=1}^T$ (graph signals) of $\mathbf{X} = [\mathbf{x}_1,\ldots,\mathbf{x}_T]$ through the GFT $\mathbf{U}^{-1}\mathbf{X}$, and in its rows $\{\mathbf{x}^i\}_{i=1}^N$ (time series) through the standard Fourier transform $\mathbf{X}\mathbf{U}_T^*$.
- We have graph-time filters that boil down to an element-wise multiplication of filter frequency response $h(\boldsymbol{\Lambda}_T, \boldsymbol{\Lambda})$ and the graph-time frequency response $\hat{\mathbf{x}}_\diamond$, i.e. a filtering operation of the form:

$$\mathbf{y}_\diamond = \mathbf{U}_\diamond h(\boldsymbol{\Lambda}_T, \boldsymbol{\Lambda})\,\mathbf{U}_\diamond^{\mathrm{H}} \mathbf{x}_\diamond. \tag{8.17}$$

Challenges

- It is difficult to know which product graph to use a priori. We can parametrize it with the parametric product graph and learn the coefficients $\{s_{ij}\}$. These parameters can be estimated during training, but the overall problem may become more challenging to handle.
- The dimensionality of the graph may arise problem in the storage and algorithmic computation, for high number of nodes N and temporal window T. In some cases, the Kronecker product properties can be exploited to alleviate these problems, by means of vectorization and parallelization. Depending on the applications, a solution is to use graph-VAR models.

8.3. Summary and Further Reading

In this chapter, we introduced time-varying aspects of GSP, a.k.a. graph-time signal processing. For in-depth mathematical details on the subject, we refer to [2]. After having motivated the necessity of such tools, (graph) autoregressive models have been considered; see [5, 9] for theoretical details on the models. The graph structure acts as a prior leading to a reduction in the number of parameters to estimate and makes the number of such parameters independent of the graph size. Next, we introduced the concepts of product-graphs as a tool to model complex datasets and to unifies different view of it (the temporal and the graph dimension). For further readings on the topic, we refer to [7], and to [13] for a signal processing view on the problem. The application of graph filters on product graph is straightforward. The introduction of the notion of parametric product graph, with learnable parameters, will be at the core of graph-time convolutional neural network architectures. See [10] for details on the parametric product graph and to [4] for its application to graph-time convolutional neural networks.

Bibliography

[1] Georgios B Giannakis, Yanning Shen, and Georgios Vasileios Karanikolas. “Topology identification and learning over graphs: Accounting for nonlinearities and dynamics”. In: *Proceedings of the IEEE* 106.5 (2018), pp. 787–807.

[2] Francesco Grassi et al. “A time-vertex signal processing framework: Scalable processing and meaningful representations for time-series on graphs”. In: *IEEE Transactions on Signal Processing* 66.3 (2017), pp. 817–829.

[3] Richard Hammack, Wilfried Imrich, and Sandi Klavžar. *Handbook of product graphs*. CRC press, 2011.

[4] Elvin Isufi and Gabriele Mazzola. “Graph-Time Convolutional Neural Networks”. In: *arXiv preprint arXiv:2103.01730* (2021).

[5] Elvin Isufi et al. “Autoregressive moving average graph filtering”. In: *IEEE Transactions on Signal Processing* 65.2 (2016), pp. 274–288.

[6] Elvin Isufi et al. “Forecasting time series with varma recursions on graphs”. In: *IEEE Transactions on Signal Processing* 67.18 (2019), pp. 4870–4885.

[7] Jure Leskovec et al. “Kronecker graphs: an approach to modeling networks.” In: *Journal of Machine Learning Research* 11.2 (2010).

[8] Helmut Lütkepohl. *New introduction to multiple time series analysis*. Springer Science & Business Media, 2005.

[9] Jonathan Mei and José MF Moura. “Signal processing on graphs: Causal modeling of unstructured data”. In: *IEEE Transactions on Signal Processing* 65.8 (2016), pp. 2077–2092.

[10] Alberto Natali, Elvin Isufi, and Geert Leus. “Forecasting Multi-Dimensional Processes Over Graphs”. In: *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2020, pp. 5575–5579.

[11] D. Romero, M. Ma, and G. B. Giannakis. “Kernel-Based Reconstruction of Graph Signals”. In: *IEEE Transactions on Signal Processing* 65.3 (2017), pp. 764–778. DOI: 10.1109/TSP.2016.2620116.

[12] Aliaksei Sandryhaila and José MF Moura. “Discrete signal processing on graphs”. In: *IEEE transactions on signal processing* 61.7 (2013), pp. 1644–1656.

[13] Aliaksei Sandryhaila and Jose MF Moura. “Big data analysis with signal processing on graphs: Representation and processing of massive data sets with irregular structure”. In: *IEEE Signal Processing Magazine* 31.5 (2014), pp. 80–90.

9

Graph-Time Neural Networks

In many applications, the data over a network can vary by the time that leads to a multivariate time series. Since the graph convolutional neural networks (GCNN) have shown good performance for processing the data over a network, we aim to develop them for time-dependent data over networks. In Chapter 8 we learned about linear graph-based models to analyze multivariate time-series. Same as other linear models, all of the discussed methods suffer from low descriptive power. So, in this chapter, we generalize the linear model for analyzing multivariate time-series into nonlinear models in a neural network manner.

The structure of this chapter is as follows. First, we make an introduction to provide a high-level view of different models. Then we go into the details of two interesting models; recursive models, and product graph models. Afterward we discuss the performance of these models on distinct tasks and compare them with baseline methods. Finally we conclude and summarize the chapter.

9.1. Introduction

Learning *multivariate temporal* data is a challenging task due to their intrinsic spatiotemporal dependencies. This problem emerges in many applications such as time series forecasting, time series classification, action recognition, and anomaly detection in time series [19, 18, 9, 20]. The spatial dependencies in the data can be represented effectively by a graph. As we discussed in previous chapters, graph convolutional neural networks (GCNNs) are powerful tools to process network data since they are non-linear models which also consider the graph structure [3]. So, in this chapter, we generalize GCNNs for temporal sequences over a graph to learn from multivariate temporal data.

Spatiotemporal graph-based models can be divided into hybrid and fused. *Hybrid* models combine distinct learning algorithms for graph-based data and temporal data. Hybrid models use GCNNs to extract higher-level spatial features and process these features by a temporal RNN or CNN [2, 5, 10, 12, 16, 17, 19]. On the other hand, *fused* models force the graph structure into conventional spatiotemporal models to jointly capture the spatiotemporal relationships [8, 14, 13, 15, 18]. Hence, in these models, the parameter matrix is replaced by the graph filter which is the core of GCNNs.

In this chapter, we discuss two categories of above-mentioned works; recursive models with GCNNs and product graph models. *Graph recursive models* adopt a conventional RNN (or its variants) and substitute the parameter matrix with a graph filter. The works [14, 15] generalized RNN this way and [13] also adds graph-based gating. *Product graph models* build a larger graph by connecting the graph replicas on all timestamps. This can be done by using the product graph as we discussed in Chapter 8. The work [6] has built the graph-time convolutional neural network (GTCNN) by this approach, and the work [18] has implemented the same method to address the skeleton-based action recognition task. In the following, we will elaborate these two categories with more details.

9.2. Models

9.2.1. Graph recursive models

Consider $\{\mathbf{x}_t\}_{t=1}^{T}$ is a multivariate sequential data where $\mathbf{x}_t \in \mathbb{R}^N$ is the N-dimensional vector at time t. A recurrent neural network can learn information from this sequential data by defining a hidden state variable $\mathbf{z}_t$.

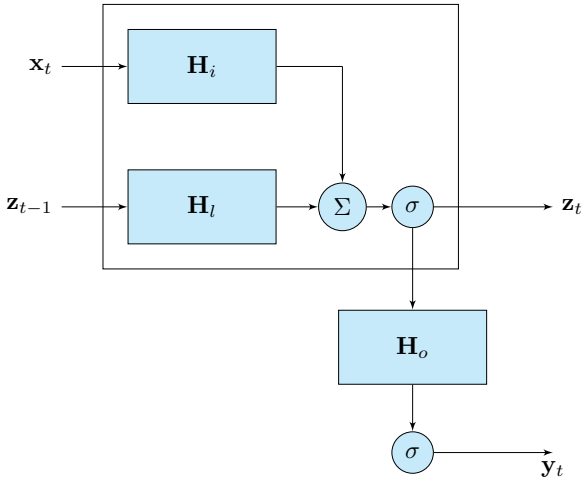


Figure 9.1: A layer of recurrent neural network.

Let the hidden states have the same dimension as the data, i.e. $\mathbf{z}_t \in \mathbb{R}^N$. Then, $\mathbf{z}_t$ can be written as a non-linear map of current input $\mathbf{x}_t$ and previous hidden state $\mathbf{z}_{t-1}$

$$\mathbf{z}_t = \sigma\left(\mathbf{H}_l \mathbf{z}_{t-1} + \mathbf{H}_i \mathbf{x}_t\right), \tag{9.1}$$

where $\sigma(\cdot)$ is a point-wise nonlinearity and $\mathbf{H}_i \in \mathbb{R}^{N\times N}$ and $\mathbf{H}_l \in \mathbb{R}^{N\times N}$ are the linear maps for input $\mathbf{x}_t$ and latent variable $\mathbf{z}_{t-1}$, respectively. The outputs $\mathbf{z}_t$ of this recursion are the higher level information of the sequence. So, we can apply another non-linear map from these outputs to the target value at each timestamp $\mathbf{y}_t \in \mathbb{R}^d$

$$\mathbf{y}_t = \sigma(\mathbf{H}_o \mathbf{z}_t), \tag{9.2}$$

where $\mathbf{H}_o \in \mathbb{R}^{d\times N}$ is the output linear map and $\sigma(\cdot)$ is again a nonlinearity. Fig. 9.1 illustrates an RNN in the block diagram form.

Based on the application, the target output $\mathscr{Y}$ can be either a label or another sequence and can be defined as a function of $\mathbf{y}_t$, i.e., $\Phi(\mathbf{y}_t)$. Given a data set $\mathscr{T} = \{(\{\mathbf{x}_t\}_{t=1}^T, \mathscr{Y})_i\}_{i=1}^M$, we can train the learnable parameters $\{\mathbf{H}_i, \mathbf{H}_l, \mathbf{H}_o\}$ to minimize a loss function $\mathscr{L}(\Phi(\mathbf{y}_t), \mathscr{Y})$. The main advantage of RNNs is that they share the parameters across the time-dimension. Hence, the number of parameters is not dependent to the sequence length so they can handle both large time sequences and sequences with different length.

Now, consider each temporal measurement $\mathbf{x}_t$ is a graph signal. We can encode the graph structure in (9.1) by replacing the linear transformations with graph filters. This model is named *Graph Recurrent Neural Network* (GRNN) and it is defined as [13]

$$\mathbf{z}_t = \sigma\left(\mathbf{H}_l(\mathbf{S}) \mathbf{z}_{t-1} + \mathbf{H}_i(\mathbf{S}) \mathbf{x}_t\right), \tag{9.3}$$

where $\mathbf{H}_i(\mathbf{S})$ and $\mathbf{H}_l(\mathbf{S})$ are graph filters, i.e. an order-K polynomial of the graph shift operator $\mathbf{S}$. Expanding the filters, we can rewrite (9.3) as

$$\mathbf{z}_t = \sigma\left(\sum_{k=0}^{K} h_{lk} \mathbf{S}^k \mathbf{z}_{t-1} + \sum_{k=0}^{K} h_{ik} \mathbf{S}^k \mathbf{x}_t\right). \tag{9.4}$$

As it follows from (9.4), the previous hidden state $\mathbf{z}_{t-1}$ is treated as a graph signal and processed by filter $\mathbf{H}_l(\mathbf{S})$, which account for the multi-hop neighboring information. Likewise, the input $\mathbf{x}_t$ is processed by the other graph filter $\mathbf{H}_i(\mathbf{S})$, which process each input signal with a filter. The output $\mathbf{y}_t$ can be computed as

$$\mathbf{y}_t = \sigma(\mathbf{H}_o(\mathbf{S}) \mathbf{z}_t) = \sigma\left(\sum_{k=0}^{K} h_{ok} \mathbf{S}^k \mathbf{z}_t\right). \tag{9.5}$$

For more clarification, Fig. 9.2 indicates the GRNN in a block diagram. By applying the graph filters the number of learnable parameters decreases significantly since we need to only train K filter coefficients

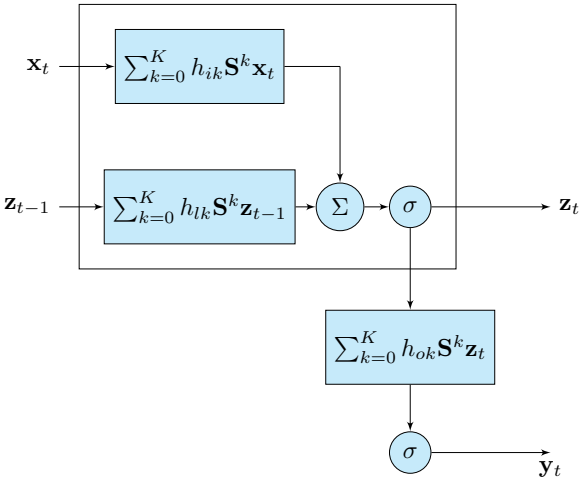
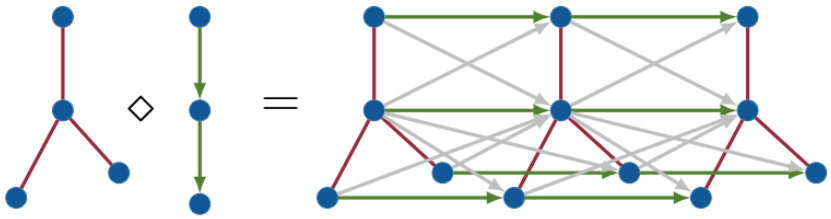


Figure 9.2: A layer of graph recurrent neural network.



Kronecker: $\mathbf{S}_{\otimes} = \mathbf{S}_T \otimes \mathbf{S}$; $\quad |\mathcal{E}_{\otimes}| = 2|\mathcal{E}_T||\mathcal{E}|$

Cartesian: $\mathbf{S}_{\times} = \mathbf{S}_T \otimes \mathbf{I}_N + \mathbf{I}_T \otimes \mathbf{S}$; $\quad |\mathcal{E}_{\times}| = T|\mathcal{E}| + N|\mathcal{E}_T|$

Strong: $\mathbf{S}_{\boxtimes} = \mathbf{S}_{\otimes} + \mathbf{S}_{\times}$ $\quad |\mathcal{E}_{\boxtimes}| = |\mathcal{E}_{\otimes}| + |\mathcal{E}_{\times}|$

Figure 9.3: Product graphs. Kronecker product has the gray edges. Cartesian has the red and green edges. Strong product has all the red, gray, and green edges. Parametric product has all the edges and also self loops if all $s_{ij} \neq 0$. Adopted from [6].

$\{\mathbf{H}_i, \mathbf{H}_l, \mathbf{H}_o\}$ instead of N^2 parameters for each linear transformation $\{\mathbf{H}_i, \mathbf{H}_l, \mathbf{H}_o\}$. Moreover, the GRNN inherits permutation equivariance and locality of graph filters.

Equations (9.4) and (9.5) concern the graph-based RNN formulation to learn from structured temporal sequences. However, as the conventional RNNs, they may suffer processing long temporal sequences; i.e., vanishing and exploding gradients. To overcome this, such methods can be extended into LSTMs, where in the latter each parameter matrix (gates, memory, state, output) is substituted by a graph filter [14, 13, 11].

9.2.2. Product graph models

RNNs are only one viewpoint for processing temporal sequences. As we discussed in Chapter 8, a multivariate time series $\mathbf{x} = [\mathbf{x}_1, \ldots, \mathbf{x}_T]$ can be vectorized and represented over the parametric product graph $\mathcal{G}_{\diamond} = \mathcal{G}_T \diamond \mathcal{G}$, where $\mathcal{G}_T = \{\mathcal{V}_T, \mathcal{E}_T\}$ and $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ are the temporal and spatial graphs, respectively. So, we can define the graph-time filter for the input $\mathbf{x}_{\diamond} = \text{vec}(\mathbf{x})$ as [6]

$$\mathbf{y}_{\diamond} = \sum_{k=0}^{K} h_k \mathbf{S}_{\diamond}^k \mathbf{x}_{\diamond} = \sum_{k=0}^{K} h_k \left(\sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\mathbf{S}_T^i \otimes \mathbf{S}^j) \right)^k \mathbf{x}_{\diamond}, \tag{9.6}$$

where $\mathbf{S}_{\diamond} = \sum_{i=0}^{1} \sum_{j=0}^{1} s_{ij} (\mathbf{S}_T^i \otimes \mathbf{S}^j)$, s_{ij} is the graph product parameter, and h_k is the filter coefficient. When we product the graphs, the spatial graph replicates T times over the time dimension, and the choice of s_{ij} and $\mathbf{S}_T$ determine the structure of the product graph, see Fig. 9.3.

Upon defining a product graph $\mathcal{G}_{\diamond}$ and a product graph signal $\mathbf{x}_{\diamond}$, we can re-use the GCNN idea to develop a neural network over the product graph following the first principles of the convolution operation. A graph-time convolutional neural network (GTCNN) is a cascaded architecture of L layers each having a graph-time convolutional filter and a point-wise nonlinearity. At layer l, we have a collection of F graph signal features

$\mathbf{x}^g_{\diamond,l-1}$ for $g = 1,\dots,F$ as the input. Each input feature $\mathbf{x}^g_{\diamond,l-1}$ is processed in parallel by a bank of F_l graph-time filters $\mathbf{H}^{fg}_l(\mathbf{S}_\diamond)$ to yield the filtered features [6]

$$\mathbf{z}^{fg}_{\diamond,l} = \mathbf{H}^{fg}_l(\mathbf{S}_\diamond)\mathbf{x}^g_{\diamond,l-1} \quad \text{for} \quad (f;g) = (1;1),\dots,(F;F). \tag{9.7}$$

Filtered features $\mathbf{z}^{fg}_{\diamond,l}$ obtained from a common input $\mathbf{x}^g_{\diamond,l-1}$ are summed to form the higher-level linear features of layer l

$$\mathbf{z}^f_{\diamond,l} = \sum_{g=1}^{F} \mathbf{z}^{fg}_{\diamond,l} = \sum_{g=1}^{F} \mathbf{H}^{fg}_l(\mathbf{S}_\diamond)\mathbf{x}^g_{\diamond,l-1} \quad \text{for} \quad f = 1,\dots,F \tag{9.8}$$

which are a collection of F graph signal features. Now, these higher-level linear features are passed to a point-wise nonlinearity to produce a collection of F_l higher-level nonlinear features $\mathbf{x}^f_{\diamond,l}$ which constitute the output of layer l,

$$\mathbf{x}^f_{\diamond,l} = \sigma_l(\mathbf{z}^f_{\diamond,l}) \quad \text{for} \quad f = 1,\dots,F_l \tag{9.9}$$

where $\sigma(\cdot)$ is the nonlinearity.

In the last layer $l = L$, we assume there is a single feature signal $F_L = 1$, which we consider the output of the GTCNN. We write this output compactly as [6]

$$\Phi(\mathbf{x}_\diamond;\mathbf{S}_\diamond;\mathscr{H}) = \sigma_L\left(\sum_{g=1}^{F} \mathbf{H}^{fg}_L(\mathbf{S}_\diamond)\mathbf{x}^g_{\diamond,L-1}\right) \tag{9.10}$$

to specify the dependence from the product graph $\mathbf{S}_\diamond$, graph-time signal $\mathbf{x}_\diamond$, and parameters set $\mathscr{H}$ defining all graph-time filters in (9.8).

We now discuss the recursive implementation of the GTCNN to provide insights on its computational complexity and scalability. If the product graph is fixed, $\mathbf{S}_\diamond$ has a sparsity of order $|\mathscr{E}_\diamond| = NT + N|\mathscr{E}_T| + T|\mathscr{E}| + |\mathscr{E}_T||\mathscr{E}|$. Computing output $\mathbf{y}_\diamond$ requires computing the shifts $\mathbf{x}^{(k)}_\diamond = \mathbf{S}^k_\diamond\mathbf{x}_\diamond$. For this, we can use the well-known recursive implementation of shifting signals over a graph [7] and write $\mathbf{x}^{(k)}_\diamond = \mathbf{S}^k_\diamond\mathbf{x}_\diamond = \mathbf{S}_\diamond\mathbf{x}^{(k-1)}_\diamond$. Hence we can obtain the output $\mathbf{y}_\diamond$ with the linear cost $\mathscr{O}(K|\mathscr{E}_\diamond|)$ as for the GCNN.

If the parameters $\{s_{ij}\}$ are to be learned, computing $\mathbf{S}_\diamond$ beforehand or using (9.6) can be unaffordable in large-scale settings since the powers of $\mathbf{S}_\diamond$ have cubic cost $\mathscr{O}((NT)^3)$. To allow scalability, we first expand all polynomials of order k and rearrange the terms to write (9.6) as

$$\mathbf{y}_\diamond = \sum_{k=0}^{K} h_k\mathbf{S}^k_\diamond\mathbf{x}_\diamond = \sum_{k=0}^{\tilde{K}}\sum_{1=0}^{\tilde{K}} h_{kl}(\mathbf{S}^l_T \otimes \mathbf{S}^k)\mathbf{x}_\diamond \tag{9.11}$$

for some orders $\bar{K}$ and $\tilde{K}$ and parameters $\{h_{kl}\}$. To compute output $\mathbf{y}_\diamond$, we need to compute all terms of the form $\mathbf{x}^{kl}_\diamond = (\mathbf{S}^l_T \otimes \mathbf{S}^k)\mathbf{x}_\diamond$. Exploiting the Kronecker product property $(\mathbf{A}\otimes\mathbf{B})(\mathbf{C}\otimes\mathbf{D}) = \mathbf{AC}\otimes\mathbf{BD}$, we can write the latter as

$$\mathbf{x}^{(k,l)}_\diamond = (\mathbf{S}_T \otimes \mathbf{I}_N)(\mathbf{I}_T \otimes \mathbf{S})(\mathbf{S}^{l-1}_T \otimes \mathbf{S}^{k-1})\mathbf{x}_\diamond. \tag{9.12}$$

Thus, we can compute $\mathbf{x}^{(k,l)}_\diamond$ recursively again as

$$\begin{aligned}\mathbf{x}^{(k,l)}_\diamond &= (\mathbf{S}_T \otimes \mathbf{I}_N)(\mathbf{I}_T \otimes \mathbf{S})\mathbf{x}^{(k-1,l-1)}_\diamond \\ &= (\mathbf{S}_T \otimes \mathbf{I}_N)\mathbf{x}^{(k,l-1)}_\diamond\end{aligned} \tag{9.13}$$

with initialization $\mathbf{x}^{(0,0)}_\diamond = \mathbf{x}_\diamond$. Recursion (9.13) implies we can compute $\mathbf{x}^{(k,l)}_\diamond$ from $\mathbf{x}^{(k-1,l-1)}_\diamond$ with a cost of $\mathscr{O}(T|\mathscr{E}| + N|\mathscr{E}_T|)$ and since we need to perform the latter for all $k \in [\bar{K}]$ and $l \in [\tilde{K}]$, we have a computational cost of $\mathscr{O}(\bar{K}\tilde{K}T|\mathscr{E}| + \bar{K}\tilde{K}N|\mathscr{E}_T|)$, which is linear in the product graph dimension.

Note also that form (9.11) improves our control on the spatiotemporal locality through orders $\bar{K}$ and $\tilde{K}$. A larger $\bar{K}$ implies more reach over the spatial graph, while a larger $\tilde{K}$ implies more reach over the temporal graph. Both borders are design choices.

9.3. Applications

Here, we illustrate the graph-time convolutional neural networks on three different applications to provide insights about its inner-working mechanisms. All results are adopted from [6]. The temporal graph $\mathbf{S}_T$ is the directed line in all the experiments.

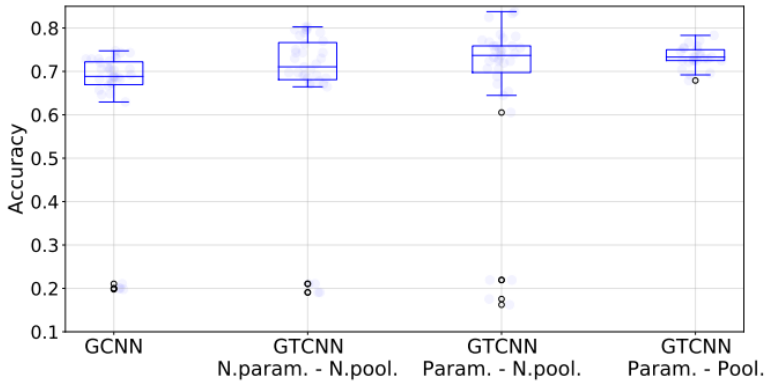


Figure 9.4: Comparison of the GCNN baseline with the non-parametric and parametric GTCNN without pooling and with the parametric GTCNN with pooling.

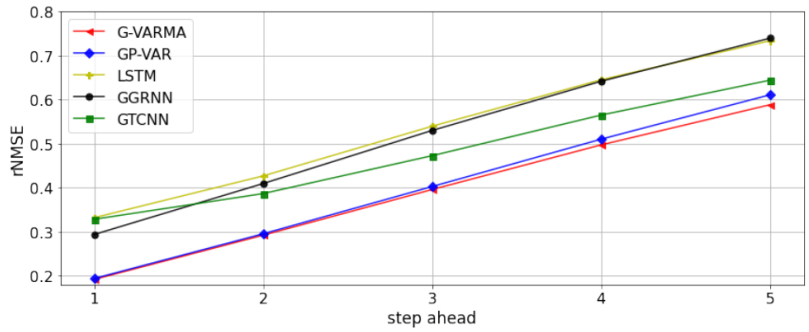


Figure 9.5: Root normalized MSE versus future prediction steps for the different methods in the Molene dataset.

9.3.1. Source localization

The task consists of finding the source of a diffusion process by observing a sequence of T graph signals $\mathbf{x}_t, \ldots, \mathbf{x}_{t+T}$ for a random time insstance t. The graph is undirected stochastic block model of $N = 100$ nodes and $C = 5$ communities. For a fair comparison with the GCNN baseline, we considered T successive signal realizations as features in the input layer. We evaluated features in $F_1, F_2 \in \{2, 4, 16, 32\}$. For the GTCNN it is considered the zero-pad pooling among layers to reduce the computational complexity.

Fig. 9.4 compares the GTCNNs with parametric and non-parametric product graphs with the baseline GCNN. We can see that accounting for the temporal domain via the sparse connectivity of the product graph improves upon GCNN solutions. Better results are achieved via the parametric product graph and by the use of pooling as evidenced by the larger median value and the smaller deviation of right-most boxplot.

9.3.2. Forecasting

We now consider the task of forecasting future values of a multivariate time-varying signal given a sequence of T past realizations. We consider the setting in [8] and considered the Molene dataset comprising 744 hourly temperature measurements across $N = 32$ stations in a region of France; and the NOAA dataset comprising 8579 hourly temperature measurements across 109 places in the continental U.S.. To train the architectures the loss function is the MSE between the one-step ahead predictions $\hat{\mathbf{x}}_{t+1}$ and the true value $\mathbf{x}_{t+1}$ regularized by the l_1-norm of all parametric product graph coefficients $\mathbf{s} = \text{vec}(\{s^{fg}_{ij,l}\})$, i.e., $\mathscr{L} = \text{MSE}(\hat{\mathbf{x}}_{t+1}, \mathbf{x}_{t+1}) + \beta\|\mathbf{s}\|_1$, where $\beta \geq 0$ is a scalar. We compared the GTCNN with: i) the linear models G-VARMA and GP-VAP [8]; ii) the gated graph-based RNN (GGRNN) [13]; iii) the LSTM.

Figs. 9.5 and 9.6 show the root normalized MSE (rNMSE) for up to five steps ahead prediction for the Molene and the NOAA dataset, respectively. For Molene dataset, we can see that all graph-based approaches achieve a lower rNMSE than the LSTM. This is because the dataset contains fewer training samples ; thus imposing an inductive bias [1] through the product graph during learning is helpful. In fact, the best performance

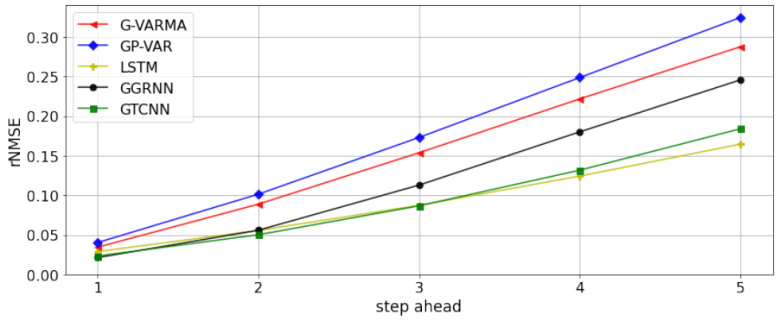


Figure 9.6: Root normalized MSE versus future prediction steps for the different methods in the NOAA dataset.

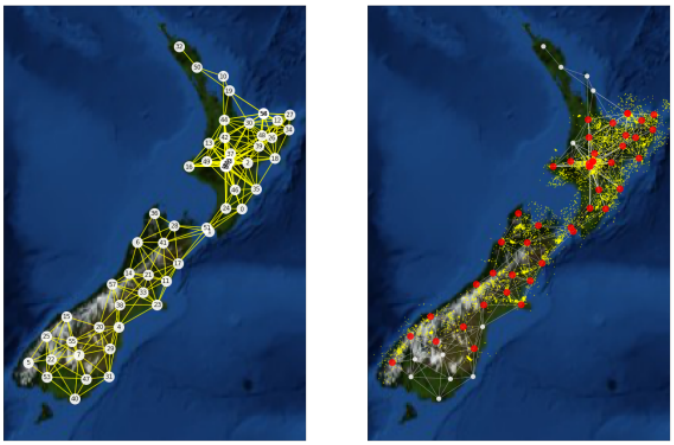


Figure 9.7: (Left) Graph structure among the seismic stations. (Right) Yellow dots are the earthquake epicenters; red nodes are stations with an assigned label; white nodes are stations without label.

is achieved by the linear graph models, while the GTCNN performs the best among the neural network alternatives. For the NOAA dataset, instead, we see the opposite trend: the neural network solutions achieve a lower rNMSE compared with the linear graph models. Since the NOAA dataset contains more training samples it allows neural networks to learn more complicated representations. The GTCNN achieves the best performance together with the LSTM, while the GGRNN suffers when predicting more than three steps. Overall, these results put the GTCNN as a valid alternative to learn representations with inductiove spatiotemporal biases when both the number training samples is limited and large.

9.3.3. Earthquake classification

Lastly, we consider an experiment to find the epicenter of earthquakes [4]. For this task we rely only on wave recordings up to 20 seconds before the strike. We built a dataset from the New Zeeland earthquake service. We considered 4,633 seismic wave recordings between 2016 and 2020 across 58 stations. We built a geometric graph of $N = 58$ nodes and an edge exists if two stations are within 170km. Fig. 9.7 illustrates the graph and the earthquake distribution. The graph signal $\mathbf{x}_t$ consists of 20 timestamps of recording in the ten seconds before the strike over all 58 stations. We compared again the GTCNN with the LSTM and the GGRNN. Here, for simplicity, we consider *one-vs-all* binary classification problem, which assigns the wave to a specific station or any of the other 44 stations.

From Fig. 9.8, we can see that, while all the methods have a very similar statistical performance, the GTCNN has higher average value compared with the other alternatives.

9.4. Summary and Further Reading

In this chapter, we discussed the GCNN solutions for analyzing multivariate time series since graphs are decent tools to model the internal dependencies in a dataset. We divided these solutions into *hybrid* and *fused* models,

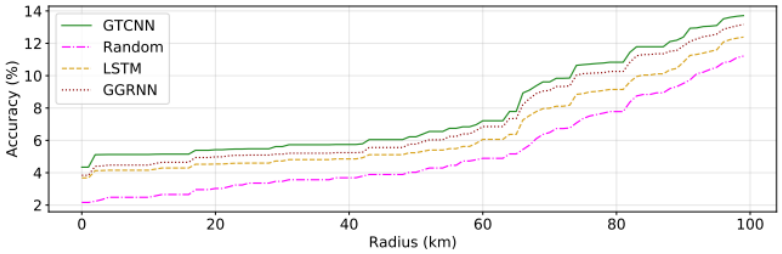


Figure 9.8: Radius-based accuracy results of the different models as a function of the distance from the correct station. I.e., a classification for an earthquake with label station i is considered correct even if assigned to a station that is within a given radius.

and focused on the latter.. These models force the graph structure into conventional models and develop new architecture to process spatiotemporal data. In other words, in these models, the parameter matrix is substituted with the graph filter. For more details, interested readers can refer to [8, 14, 13, 15, 18].

Finally, we elaborated two categories among the fused models: recursive models, and product graph models. Recursive models adopt a conventional RNN and force the parameter matrix to be a polynomial in graph shift operator. So, the linear transformation converts into the graph filter and the GRNN emerges [14, 13]. Product graph models assume two spatial and temporal graphs and build a larger graph by multiplying these two graphs [6]. Hence, the product graph is now responsible for both spatial and temporal dependencies simultaneously and one can train this GTCNN to capture spatiotemporal patterns.

Bibliography

[1] P. W. Battaglia et al. “Relational inductive biases, deep learning, and graph networks”. In: *arXiv preprint arXiv:1806.01261* (2018).

[2] D. Chai, L. Wang, and Q. Yang. “Bike flow prediction with multi-graph convolutional networks”. In: *Proceedings of the 26th ACM SIGSPATIAL international conference on advances in geographic information systems*. 2018, pp. 397–400.

[3] F. Gama et al. “Graphs, convolutions, and neural networks: From graph filters to graph neural networks”. In: *IEEE Signal Processing Magazine* 37.6 (2020), pp. 128–138.

[4] R. J. Geller. “Earthquake prediction: a critical review”. In: *Geophysical Journal International* 131.3 (1997), pp. 425–450.

[5] S. Guo et al. “Attention based spatial-temporal graph convolutional networks for traffic flow forecasting”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 922–929.

[6] E. Isufi and G. Mazzola. “Graph-Time Convolutional Neural Networks”. In: *IEEE Data Science and Learning Workshop* (2021).

[7] E. Isufi et al. “Autoregressive moving average graph filtering”. In: *IEEE Transactions on Signal Processing* 65.2 (2016), pp. 274–288.

[8] E. Isufi et al. “Forecasting time series with varma recursions on graphs”. In: *IEEE Transactions on Signal Processing* 67.18 (2019), pp. 4870–4885.

[9] M.W. Kadous et al. *Temporal classification: Extending the classification paradigm to multivariate time series*. Citeseer, 2002.

[10] M. Khodayar and J. Wang. “Spatio-temporal graph deep neural network for short-term wind speed forecasting”. In: *IEEE Transactions on Sustainable Energy* 10.2 (2018), pp. 670–681.

[11] Y. Li et al. “Diffusion convolutional recurrent neural network: Data-driven traffic forecasting”. In: *arXiv preprint arXiv:1707.01926* (2017).

[12] F. Manessi, A. Rozza, and M. Manzo. “Dynamic graph convolutional networks”. In: *Pattern Recognition* 97 (2020), p. 107000.

[13] L. Ruiz, F. Gamao, and A. Ribeiro. “Gated graph recurrent neural networks”. In: *IEEE Transactions on Signal Processing* 68 (2020), pp. 6303–6318.

[14] Y. Seo et al. “Structured sequence modeling with graph convolutional recurrent networks”. In: *International Conference on Neural Information Processing*. Springer. 2018, pp. 362–373.

[15] C. Si et al. “An attention enhanced graph convolutional lstm network for skeleton-based action recognition”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 1227–1236.

[16] Y. Sun et al. “Constructing geographic and long-term temporal graph for traffic forecasting”. In: *arXiv preprint arXiv:2004.10958* (2020).

[17] X. Wu et al. “Graph wavenet for deep spatial-temporal graph modeling”. In: *arXiv preprint arXiv:1906.00121* (2019).

[18] S. Yan, Y. Xiong, and D. Lin. “Spatial temporal graph convolutional networks for skeleton-based action recognition”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 2018.

[19] B. Yu, H. Yin, and Z. Zhu. “Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting”. In: *arXiv preprint arXiv:1709.04875* (2017).

[20] C. Zhang et al. “A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 2019, pp. 1409–1416.

Appendix A: Linear Algebra

Linear Algebra is fundamental to understand machine learning. In this chapter, we review some basic concepts of linear algebra that will be useful in the upcoming chapters.

.1. Definitions

A column vector $\mathbf{a}$ of dimension $N \times 1$ is a collection of N real numbers in the N-dimensional Euclidean space. A matrix $\mathbf{A}$ of dimensions $M \times N$ is a collection of MN real numbers, arranged in M rows and N columns, where the (i, j)th element of $\mathbf{A}$ is A_{ij}. Matrix $\mathbf{A}$ is a collection of N M-dimensional column vectors, i.e. $\mathbf{A} = [\mathbf{a}_1, \ldots, \mathbf{a}_M]$. The transpose of $\mathbf{A}$ is denoted by $\mathbf{A}^\top \in \mathbb{R}^{N \times M}$ and converts the columns of $\mathbf{A}$ into rows. For a complex matrix $\mathbf{A} \in \mathbb{C}^{M \times N}$, its Hermitian $\mathbf{A}^H$ is a combination of its transpose and element-wise complex conjugate. A real matrix $\mathbf{A}$ is symmetric if $\mathbf{A} = \mathbf{A}^\top$. In this section, we look at some popular products between vectors and matrices, some special matrices and norms.

.1.1. Vector Norms

- `Vector norm`: The norm of a vector $\mathbf{a} \in \mathbb{R}^N$ is denoted by $||\mathbf{a}||$. The norm is a mapping from $\mathbb{R}^N$ to $\mathbb{R}$ which obeys the following properties:
  - $||\mathbf{a}|| \geq 0$,
  - $||c\mathbf{a}|| = |c|||\mathbf{a}||$ for a scalar c;
  - $||\alpha\mathbf{a} + \beta\mathbf{b}|| \leq |\alpha|||\mathbf{a}|| + |\beta|||\mathbf{b}||$.

  Different norms are possible, each having its own interpretation and use.
- `Vector p-norm`: The p-norm of vector $\mathbf{a} = [a_1, \ldots, a_N]^\top$ is defined as

$$||\mathbf{a}||_p = \left[\sum_{n=1}^{N} |a_n|^p \right]^{\frac{1}{p}} \text{ for } p \geq 1. \tag{14}$$

  For $0 \leq p \leq 1$, the p-norm is often called the pseudo-norm because it no longer satisfies the three properties.
- l_0, l_1, l_2, `inf norm`:
  - l_0 norm: The l_0 norm of $\mathbf{a}$, while not obeying the norm rules, is the number of non zero elements in $\mathbf{a}$; i.e.,

$$||\mathbf{a}||_0 = \sum_{n=1}^{N} \mathbb{I}(a_n \neq 0), \tag{15}$$

    where $\mathbb{I}(\cdot)$ is the indicator function. The l_0 norm is also called the cardinality of the vector.
  - l_1 norm: The l_1 norm of $\mathbf{a}$ corresponds to $p = 1$. It is the sum of the absolute values of the elements in $\mathbf{a}$; i.e.,

$$||\mathbf{a}||_1 = \sum_{n=1}^{N} |a_n|. \tag{16}$$

  - l_2 norm: The l_2 norm of $\mathbf{a}$ is a measure of the energy of $\mathbf{a}$; i.e.,

$$||\mathbf{a}||_2 = \sqrt{\sum_{n=1}^{N} a_n^2}. \tag{17}$$

    It is used widely to define distance between vectors in the Euclidean space, as we shall see shortly.

– l_∞ norm: The l_∞ norm is the largest absolute value among the elements of $\mathbf{a}$, i.e.,

$$||\mathbf{a}||_\infty = \max_n |a_n| \tag{18}$$

– `Vector distance`: The distance between two vectors $\mathbf{a}$ and $\mathbf{b}$ is

$$d(\mathbf{a},\mathbf{b}) = ||\mathbf{a}-\mathbf{b}||_2 = \sqrt{(\mathbf{a}-\mathbf{b})^\top(\mathbf{a}-\mathbf{b})}. \tag{19}$$

.1.2. Common Linear Algebra Products

We will now see common operations in the form of products between vectors and matrices.

- `Inner product`: The inner (dot) product between two vectors $\mathbf{a}$ and $\mathbf{b}$ of dimensions $N \times 1$ is defined as

$$\mathbf{a}^\top\mathbf{b} = \mathbf{b}^\top\mathbf{a} = \sum_{n=1}^{N} a_n b_n. \tag{20}$$

  Two vectors are orthogonal if their inner product is zero i.e., $\mathbf{a}$ and $\mathbf{b}$ are perpendicular to each other. When $\mathbf{a} = \mathbf{b}$, the inner product is the squared norm of $\mathbf{a}$ i.e., $||\mathbf{a}||_2^2 = \sum_{n=1}^{N} a_n^2$.

- `Projection`: The projection of a vector $\mathbf{a}$ onto another vector $\mathbf{b}$ of same dimension is

$$\Pi_{\mathbf{b}}(\mathbf{a}) = \frac{\mathbf{a}^\top\mathbf{b}}{||\mathbf{b}||_2}\mathbf{b}. \tag{21}$$

  The projection operation tells how aligned are vectors $\mathbf{a}$ and $\mathbf{b}$. If we have perfect alignment $\mathbf{a} = \mathbf{b}$, we have the projection of the vector onto itself, yielding $\mathbf{a}$. If the two vectors are orthogonal we have $\Pi_{\mathbf{b}}(\mathbf{a}) = 0$, which indicates the vectors are not aligned at all.

- `Matrix-vector product`: The matrix-vector product of matrix $\mathbf{A} \in \mathbb{R}^{M\times N}$ and vector $\mathbf{b} \in \mathbb{R}^{N\times 1}$ yields a vector $\mathbf{c} = \mathbf{Ab}$. This is a linear combination of the columns of $\mathbf{A}$; i.e. $\mathbf{c} = \mathbf{Ab} = \sum_{n=1}^{N} b_n \mathbf{a}_n$.

- `Matrix-matrix product`: The matrix-matrix multiplication takes matrices $\mathbf{A} \in \mathbb{R}^{M\times N}$ and $\mathbf{B} \in \mathbb{R}^{N\times K}$ and outputs a matrix $\mathbf{C} = \mathbf{AB}$ of size $M \times K$ with (i,j)th entry $C_{ij} = \sum_{n=1}^{N} A_{in}B_{nj}$.

- `Matrix Kronecker product`: For two matrices $\mathbf{A} \in \mathbb{R}^{M\times N}$ and $\mathbf{B} \in \mathbb{R}^{P\times Q}$, their matrix Kronecker product outputs the block matrix $\mathbf{C}$ of size $MP \times NQ$

$$\mathbf{C} = \mathbf{A}\otimes\mathbf{B} = \begin{pmatrix} A_{11}\mathbf{B} & \cdots & A_{1N}\mathbf{B} \\ \vdots & \ddots & \vdots \\ A_{M1}\mathbf{B} & \cdots & A_{MN}\mathbf{B} \end{pmatrix} \tag{22}$$

  Notice that $\mathbf{A}\otimes\mathbf{B} = \mathbf{B}\otimes\mathbf{A}$.

- `Matrix Cartesian product`: For two square matrices $\mathbf{A} \in \mathbb{R}^{M\times M}$ and $\mathbf{B} \in \mathbb{R}^{N\times N}$, the matrix Cartesian product is defined as

$$\mathbf{C} = \mathbf{C} = \mathbf{A}\times\mathbf{B} = \mathbf{A}\otimes\mathbf{I}_N + \mathbf{I}_M\otimes\mathbf{B} \tag{23}$$

  where $\mathbf{I}_N$ is the identity matrix of size N.

- `Matrix Hadamard product`: For two matrices $\mathbf{A}$ and $\mathbf{B}$ of dimensions $M \times N$, the Hadamard product $\mathbf{C} = \mathbf{A}\circ\mathbf{B}$ is their element-wise product with $C_{ij} = A_{ij}B_{ij}$.

- `Matrix Strong product`: For two square matrices $\mathbf{A} \in \mathbb{R}^{M\times M}$ and $\mathbf{B} \in \mathbb{R}^{N\times N}$, the matrix strong product is defined as

$$\mathbf{C} = \mathbf{C} = \mathbf{A}\boxtimes\mathbf{B} = \mathbf{A}\otimes\mathbf{B} + \mathbf{A}\otimes\mathbf{I}_N + \mathbf{I}_M\otimes\mathbf{B}. \tag{24}$$

  The strong produt is the combination of the Kronecker and Cartesian products.

.1.3. Special Matrices

- `Diagonal matrix`: A diagonal matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ has non-zero values only along its main diagonal. It has $A_{ij} \neq 0$ if and only if $i = j$ and zero otherwise.
- `Diagonally dominant matrix`: A diagonally dominant matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a square matrix such that for every row i, the absolute value of the diagonal element is greater than or equal to the sum of the absolute value of the non-diagonal elements; i.e.,

$$|A_{ii}| \geq \sum_{j \neq i} |A_{ij}|, \quad n = 1, \ldots, N \tag{25}$$

- `Toeplitz matrix`: A Toeplitz matrix $\mathbf{A}$ has constant values along each diagonal

$$\mathbf{A} = \begin{pmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & \vdots \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{n-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{pmatrix} \tag{26}$$

. Notice that we need to know the first column and first row of a Toeplitz matrix to characterize it.

- `Circulant matrix`: A circulant matrix $\mathbf{A}$ is a matrix in which each column is rotated by one element around the matrix relative to the preceding column; i.e.,

$$\mathbf{A} = \begin{pmatrix} c_0 & c_{n-1} & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ c_{n-2} & & \ddots & c_0 & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & c_1 & c_0 \end{pmatrix} \tag{27}$$

A circulant matrix is Toeplitz and it is characterized by knowing its first column.

- `Orthogonal matrix`: An orthogonal matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is a real square matrix with its columns and rows forming two sets of orthogonal vectors. For such a matrix, $\mathbf{A}^\top \mathbf{A}$ and $\mathbf{A}\mathbf{A}^\top$ are diagonal matrices

$$\mathbf{A}^\top \mathbf{A} = \text{diag}(||\mathbf{a}_1||_2^2, \mathbf{a}_2||_2^2, \ldots, ||\mathbf{a}_N||_2^2) \tag{28}$$

where $\mathbf{a}_k$ is the k-th column of $\mathbf{A}$. Also,

$$\mathbf{A}\mathbf{A}^\top = \text{diag}(||\mathbf{a}_1^\top||_2^2, ||\mathbf{a}_2^\top||_2^2, \ldots, ||\mathbf{a}_N^\top||_N^2) \tag{29}$$

where $\mathbf{a}_k^\top$ is the k-th row of $\mathbf{A}$.

- `Orthonormal matrix`: An orthonormal matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is an orthogonal matrix with all of its columns and rows having unit norm; i.e., $\mathbf{A}^\top \mathbf{A} = \mathbf{A}\mathbf{A}^\top = \mathbf{I}_N$.
- `Inverse of a matrix`: The inverse of a square matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is the matrix $\mathbf{A}^{-1} \in \mathbb{R}^{N \times N}$ such that

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{A}^{-1}\mathbf{A} = \mathbf{I}_N \tag{30}$$

.2. Linear Independence, Rank, and Related Concepts

In this section, we review the rank of a matrix, the inverse, the pseudo-inverse of a matrix.

- `Linear independence`: A set of vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is linearly independent when

$$\sum_{n=1}^{N} \alpha_n \mathbf{x}_n = \mathbf{0}_N \tag{31}$$

if and only if all coefficients α_n are zero, i.e., $\boldsymbol{\alpha} = [\alpha_1, \ldots, \alpha_N]^\top = \mathbf{0}_N$.

- `Linear dependence`: A set of vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is linearly dependent when

$$\sum_{n=1}^{N} \alpha_n \mathbf{x}_n = \mathbf{0}_N \tag{32}$$

  for at least one $\alpha_n \neq 0$.

- `Span`: The span of a set of vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ is the vector space covered by all possible linear combinations of the vectors. If the N vectors are independent, the dimensionality of the span is N. If the vectors are dependent and only M are independent, the dimensionality of the span is M.

- `Basis:` A basis in $\mathbb{R}^N$ is any set of N independent vectors $\mathbf{x}_1, \ldots, \mathbf{x}_N$ that spans $\mathbb{R}^N$.

- `Rank:` The rank of an matrix $\mathbf{A}$ is the dimensionality of the span (dim-span) of its columns or rows. The column (row) rank of $\mathbf{A}$ is the dim-span of the set of its column (row) vectors. The rank of an $M \times N$ matrix satisfies

$$\text{rank}(\mathbf{A}) \leq \min(M, N) \tag{33}$$

- `Null space`: The null space of an $M \times N$ matrix $\mathbf{A}$ is the vector space comprising all $N \times 1$ vectors $\mathbf{b}$ such that

$$\mathbf{A}\mathbf{b} = \mathbf{0}_M. \tag{34}$$

  That is, the null space is the maximum space spanned by any set of linearly independent vectors, each of which satisfies condition (34).

- `Rank deficient matrix`: A matrix $\mathbf{A}$ is rank deficient when it is neither full column rank nor full row rank.

- `Rank criterion for inverse`: The inverse of a matrix $\mathbf{A}$ exists if and only if $\text{rank}(\mathbf{A}) = N$. This means the columns and rows of $\mathbf{A}$ span the N dimensional space.

- `Pseudo-inverse`: For an $M \times N$ matrix $\mathbf{A}$, the inverse is replaced with the psuedo-inverse operator $\mathbf{A}^\dagger$ defined as

$$\begin{aligned} \mathbf{A}^\dagger \mathbf{A} &= \mathbf{I}_N, \quad \text{if } M \geq N \\ \mathbf{A}\mathbf{A}^\dagger &= \mathbf{I}_M, \quad \text{if } M \leq N. \end{aligned} \tag{35}$$

.3. Matrix Spectra

In this section, we shall look at the spectrum of several matrices and related quantities, like the determinant and the rank. We will then look at matrix norms and at semidefinite matrices.

.3.1. Determinant, Eigenvectors and Eigenvalues

- `Determinant`: The determinant of a square matrix $\mathbf{A}$, denoted as $\det(\mathbf{A})$ or $|\mathbf{A}|$ is an operator mapping $\mathbf{A}$ to a scalar.

- `Eigenvalues`: The eigenvalues of an $N \times N$ matrix $\mathbf{A}$ are a set of N values $\lambda(\mathbf{A}) = \{\lambda_1, \ldots, \lambda_N\}$ associated with $\mathbf{A}$ and obtained as the roots of the characteristic polynomial of degree N

$$\det(\mathbf{A} - \lambda \mathbf{I}) = c \prod_{n=1}^{N} (\lambda - \lambda_n) = 0 \tag{36}$$

  where c is a constant. The N roots of the polynomial in (36) are the eigenvalues $\lambda_1, \ldots, \lambda_N$. The set of eigenvalues of a matrix forms the spectrum.

- `Eigenvectors`: An eigenvector $\mathbf{v}$ of $\mathbf{A}$ associated to one of its eigenvalues $\lambda \in \lambda(\mathbf{A})$ satisfies

$$\mathbf{A}\mathbf{v} = \lambda \mathbf{v}. \tag{37}$$

  There exists N eigenvectors $[\mathbf{v}_1, \ldots, \mathbf{v}_N]$ corresponding to the N eigenvalues $[\lambda_1, \ldots, \lambda_N]$ respectively. When $\mathbf{A}$ is multiplied with $\mathbf{v}$, the output is a vector in the same direction as $\mathbf{v}$ scaled by a scalar λ. An eigenvalue $\lambda_n = 0$ in (37) means the corresponding eigenvector produces the zero vector when multiplied with $\mathbf{A}$.

- `Eigendecomposition`: For a matrix $\mathbf{A}$ with eigenvalues $\{\lambda_1,\ldots,\lambda_N\}$ and eigenvectors $\{\mathbf{v}_1,\ldots,\mathbf{v}_N\}$, we can write $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1} \tag{38}$$

  where $\mathbf{V} = [\mathbf{v}_1,\ldots,\mathbf{v}_N]$ and $\mathbf{\Lambda} = \text{diag}(\lambda_1,\ldots,\lambda_N)$ is a diagonal matrix with the eigenvalues along its diagonal.

- `Eigenspace`: The space spanned by the eigenvectors of $\mathbf{A}$ [cf the columns of $\mathbf{V}$ in (38)] is called the eigenspace of $\mathbf{A}$. When $\mathbf{A}$ is invertible, its eigenvectors span the N dimensional space.

  Often, we need to project vector $\mathbf{a} \in \mathbb{R}^N$ onto a subspace contained within the eigenspace. A typical example is the principal component analysis (PCA) used to project feature vectors onto the eigenspace of the covariance matrix [4]. The projection is done on the eigenvectors corresponding to the maximum eigenvalues. For example if vector $\mathbf{x} \in \mathbb{R}^N$ has to be reduced to dimension M, the reduced feature $\mathbf{x}^{'}$ via PCA is obtained as

$$\mathbf{x}^{'} = \mathbf{V}_M^{\top}\mathbf{x} \tag{39}$$

  where $\mathbf{V}_M$ is an $N \times M$ matrix containing the eigenvectors corresponding to the M largest eigenvalues of the covariance matrix $\mathbf{\Sigma} = \mathbf{V}\boldsymbol{\lambda}\mathbf{V}^{\top}$. The matrix $\mathbf{V}_M^{\top}$ is the PCA projection matrix.

- `Singular value decomposition`: For non-square matrices, the singular value decomposition (SVD) expresses the rows and columns of a matrix in terms of a pair of orthogonal bases. Stated mathematically, the SVD of an $M \times N$ matrix $\mathbf{A}$ is

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^{T} \tag{40}$$

  where the columns of $\mathbf{U} = [\mathbf{u}_1,\ldots,\mathbf{u}_M]$ represent an orthogonal basis over $\mathbb{R}^M$ of the columns of $\mathbf{A}$, the columns of $\mathbf{V} = [\mathbf{v}_1,\ldots,\mathbf{v}_N]$ represent an orthogonal basis over $\mathbb{R}^N$ of the rows of $\mathbf{A}$. The M diagonal elements of $\mathbf{\Sigma} = \text{diag}(\sigma_1,\ldots,\sigma_M)$ denote the singular values. The singular values satisfy

$$\sigma_m \geq 0, \quad m = 1,\ldots,M. \tag{41}$$

- `Matrix norms`: Like we saw with vectors, matrices have their own norms which obey similar properties. Some popular matrix norms are:

  – Spectral norm: The Spectral norm of matrix $\mathbf{A} \in \mathbb{R}^{M\times N}$ is defined as

$$||\mathbf{A}||_2 = \max_{n=\{1,\ldots,N\}} \sqrt{\lambda_n(\mathbf{A}^H\mathbf{A})} \tag{42}$$

  – Nuclear Norm: The Nuclear norm of $\mathbf{A} \in \mathbb{R}^{M\times N}$ is

$$||\mathbf{A}||_\star = \sum_{m=1}^{M} \sigma_m \tag{43}$$

    that is the sum of the M singular values of $\mathbf{A}$. If matrix $\mathbf{A}$ is square with dimensions $N \times N$ and enjoys an eigendecomposition [cf. (9)] the nuclear norm becomes

$$||\mathbf{A}||_\star = \sum_{n=1}^{N} \lambda_n^2. \tag{44}$$

  – Frobenius Norm: The Frobenius Norm of $\mathbf{A} \in \mathbb{R}^{M\times N}$ is defined as

$$||\mathbf{A}||_F = \sqrt{\sum_{m=1}^{M}\sum_{n=1}^{N} A_{mn}^2}. \tag{45}$$

    It treats $\mathbf{A}$ as a vector and obtains its l_2 norm, i.e., it is the sum of all squared elements of $\mathbf{A}$. The Frobenius norm is often used as a measure of the energy of the elements of $\mathbf{A}$.

.3.2. Positive Semidefiniteness and Particular Eigendecompositions

- `Quadratic form`: The quadratic form of a real square matrix $\mathbf{A}$ is $\mathbf{x}^\top\mathbf{A}\mathbf{x}$, for any vector $\mathbf{x}$ of appropriate dimensions.
- `Positive semidefinite`: A positive semidefinite matrix $\mathbf{A}$ is such that its quadratic form satisfies

$$\mathbf{x}^\top\mathbf{A}\mathbf{x} \geq 0 \quad \text{for all } \mathbf{x} \neq \mathbf{0}_N \tag{46}$$

  . All eigenvalues of a positive semidefinite matrix satisfy $\lambda(\mathbf{A}) \geq 0$.
- `Positive definite`: A positive definite matrix $\mathbf{A}$ is such that its quadratic form satisfies

$$\mathbf{x}^\top\mathbf{A}\mathbf{x} > 0 \quad \text{for all } \mathbf{x} \neq \mathbf{0}_N \tag{47}$$

  . All eigenvalues of a positive definite matrix satisfy $\lambda(\mathbf{A}) > 0$.
- `Symmetric matrix`: Symmetric matrices have real eigenvalues and orthonormal eigenvectors i.e., $\mathbf{V}^\top\mathbf{V} = \mathbf{V}\mathbf{V}^\top = \mathbf{I}$, . This implies $\mathbf{V}^\top = \mathbf{V}^{-1}$ and the eigendecompostion can be written as

$$\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^\top. \tag{48}$$

- `Eigenvalues and eigenvectors of the inverse`: The inverse of $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$ has the eigendecomposition

$$\mathbf{A}^{-1} = \mathbf{V}\mathbf{\Lambda}^{-1}\mathbf{V}^{-1}. \tag{49}$$

  From this we can see that $\mathbf{A}^{-1}$ shares the same eigenvectors as $\mathbf{A}$, but the eigenvalues are inverted. If λ_n is an eigenvalue of $\mathbf{A}$, then the corresponding eigenvalue of $\mathbf{A}^{-1}$ is λ_n^{-1}.
- `Eigendecomposition of a circulant matrix`: The circulant matrix $\mathbf{A}$ of dimensions $N \times N$ has the eigendecomposition

$$\mathbf{A} = \mathbf{F}^{-1}\mathbf{\Lambda}\mathbf{F} \tag{50}$$

  where $\mathbf{F}$ is the discrete Fourier transform matrix and has the (m,n)th entry $F_{mn} = \exp(\frac{j2\pi mn}{N})$ with $j = \sqrt{-1}$. The diagonals of $\mathbf{\Lambda}$ are its eigenvalues. with entry $\lambda_n = \exp(-j\frac{2\pi(n-1)}{N})$.
- `Eigendecomposition of a power matrix`: When $\mathbf{A}$ has eigendecomposition $\mathbf{A} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{-1}$, the kth power of $\mathbf{A}$ has the eigendecomposition

$$\mathbf{A}^k = \mathbf{V}\mathbf{\Lambda}^k\mathbf{V}^{-1}. \tag{51}$$

  The kth power of $\mathbf{A}$ has the same eigenvectors as $\mathbf{A}$, whereas the eigenvalues are the kth power of the eigenvalues of $\mathbf{A}$.
- `Eigendecomposition of product matrices`: Consider matrices $\mathbf{A} \in \mathbb{R}^{M\times M}$ and $\mathbf{B} \in \mathbb{R}^{N\times N}$ with eigendecompositions $\mathbf{A} = \mathbf{V}_A\mathbf{\Lambda}_A\mathbf{V}_A^{-1}$ and $\mathbf{B} = \mathbf{V}_B\mathbf{\Lambda}_B\mathbf{V}_B^{-1}$.
  - Kronecker product: The Kronecker product between $\mathbf{A}$ and $\mathbf{B}$ has the eigendecomposition

$$\mathbf{A}\otimes\mathbf{B} = \mathbf{V}(\mathbf{\Lambda}_A\otimes\mathbf{\Lambda}_B)\mathbf{V}^{-1} \tag{52}$$

    where $\mathbf{V} = \mathbf{V}_A\otimes\mathbf{V}_B$ is the Kronecker product of the eigenvectors. The eigenvalues of the Kronecker product matrix are the Kronecker product of the eigenvalues of $\mathbf{A}$ and $\mathbf{B}$.
  - Cartesian product: The Cartesian product between $\mathbf{A}$ and $\mathbf{B}$ has the eigendecomposition

$$\mathbf{A}\otimes\mathbf{B} = \mathbf{V}(\mathbf{\Lambda}_A\otimes\mathbf{I}_N + \mathbf{I}_M\otimes\mathbf{\Lambda}_B)\mathbf{V}^{-1} \tag{53}$$

    where $\mathbf{V} = \mathbf{V}_A\otimes\mathbf{V}_B$. The eigenvalue matrix is the Cartesian product between the two eigenvalue matrices.
  - Strong product: The Strong product between $\mathbf{A}$ and $\mathbf{B}$ has the eigendecomposition

$$\mathbf{A}\boxtimes\mathbf{B} = \mathbf{V}(\mathbf{\Lambda}_A\otimes\mathbf{\Lambda}_B + \mathbf{\Lambda}_A\otimes\mathbf{I}_N + \mathbf{I}_M\otimes\mathbf{\Lambda}_B)\mathbf{V}^{-1} \tag{54}$$

    where $\mathbf{V} = \mathbf{V}_A\otimes\mathbf{V}_B$. The eigenvalue matrix is the strong product between the two eigenvalue matrices.

- `Exponential matrix`: The exponential matrix is denoted as $e^{\mathbf{A}}$ and is expressed with the matrix power series

$$e^{\mathbf{A}} = \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{A}^k. \tag{55}$$

The eigendecomposition of the exponential matrix is

$$e^{\mathbf{A}} = \mathbf{V}\Big(\sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{\Lambda}^k\Big)\mathbf{V}^{-1} = \mathbf{V}e^{\mathbf{\Lambda}}\mathbf{V}^{-1}. \tag{56}$$

The exponential matrix of $\mathbf{A}$ has the same set of eigenvectors as $\mathbf{A}$, whereas the eigenvalues of the exponential matrix are the exponential of the eigenvalues of $\mathbf{A}$.

.4. Linear System of Equations

In this section, we look at linear systems of equations. We look at overdetermined and underdetermined systems and how the respective solutions are obtained. We also look at the role of the rank of the system matrix. Lastly, we provide an example.

- `Linear system of equations`: A linear system of equations [1] is described as

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{57}$$

where the system matrix $\mathbf{A}$ has dimensions $M \times N$, vector $\mathbf{x}$ is of dimension $N \times 1$, and vector $\mathbf{b}$ has dimensions $M \times 1$. A typical goal for a linear systems of equations is: given $\mathbf{b}$ (the measurements or equations) and $\mathbf{A}$ (the system behavior), find vector $\mathbf{x}$. The solution to find $\mathbf{x}$ highly depends on the properties of matrix $\mathbf{A}$.

- `Overdetermined systems`: An overdetermined system has the number of equations M (dimensions of $\mathbf{b}$) greater than the number of variables N (dimensions of $\mathbf{x}$). The matrix $\mathbf{A}$ for such a system is called a *tall* matrix. In such a case, an exact solution may not exist. When it does exist, the solution is given as

$$\mathbf{x}^\star = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top\mathbf{b} = \mathbf{A}^\dagger\mathbf{b}. \tag{58}$$

The matrix $\mathbf{A}^\dagger = (\mathbf{A}^\top\mathbf{A})^{-1}\mathbf{A}^\top$ is the pseudo-inverse of the overdetermined matrix $\mathbf{A}$, provided $(\mathbf{A}^\top\mathbf{A})^{-1}$ exists. For the latter to happen, $\mathbf{A}$ must have full column rank (the columns span $\mathbb{R}^N$).

- `Underdetermined systems`: An underdetermined system has the number of equations M (dimensions of $\mathbf{b}$) lower than the number of variables N (dimensions of $\mathbf{x}$). The matrix $\mathbf{A}$ for such a system is called a *fat* matrix.

When $\mathbf{A}$ is underdetermined, infinite solutions exist. To limit the set of candidate solutions, a minimum norm constraint is imposed and the solution is given as

$$\mathbf{x}^\star = \mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)^{-1}\mathbf{b} = \mathbf{A}^\dagger\mathbf{b}. \tag{59}$$

This is called the minimum norm solution. The matrix $\mathbf{A}^\top(\mathbf{A}\mathbf{A}^\top)^{-1}$ is the pseudo-inverse of the underdetermined $\mathbf{A}$, provided $(\mathbf{A}\mathbf{A}^\top)^{-1}$ exists. For the latter to happen, $(\mathbf{A}^\top\mathbf{A})^{-1}$ must exist and that is possible when $\mathbf{A}$ has full row rank (the rows span $\mathbb{R}^M$).

.5. Summary and Further Reading

Interested readers are encouraged to read [3] for basic concepts. Details about linear systems and optimization surrounding them are found in [1]. For an almost exhaustive list of matrix properties, functions, and derivatives, information is available in [2].

Bibliography

[1] Stephen Boyd and Lieven Vandenberghe. *Introduction to applied linear algebra: vectors, matrices, and least squares*. Cambridge university press, 2018.

[2] Anncatrine Luisa Petersen. "Liposomal delivery of radionuclides for cancer diagnostics and radiotherapy: From material development to in vivo applications using positron emission tomography (PET) imaging". In: (2012).

[3] Gilbert Strang. *Introduction to linear algebra*. Vol. 3. Wellesley-Cambridge Press Wellesley, MA, 1993.

[4] Svante Wold, Kim Esbensen, and Paul Geladi. "Principal component analysis". In: *Chemometrics and intelligent laboratory systems* 2.1-3 (1987), pp. 37–52.

Appendix B: Convex Optimization

Throughout this handout, we will deal with *functions*, which are operators that map each element of a set to one element of another set. Think for instance of the function $f(x) = x$ which maps real numbers onto themselves. In this introductory chapter we will work with real-valued functions, and recall some of the basic definitions that will be used in the subsequent chapters. While the output of a function will be mainly considered a scalar, it *argument* may be a scalar, a vector, or a matrix.

- **Functions.** Given two sets $\mathscr{X}$ and $\mathscr{Y}$, a function $f : \mathscr{X} \to \mathscr{Y}$ is a relation that maps each element $x \in \mathscr{X}$ to one element $y \in \mathscr{Y}$. We also say that $\mathscr{X}$ and $\mathscr{Y}$ are the *domain* and the *codomain* of $f(\cdot)$, respectively.

- **Level sets.** A level set of a function $f(\mathbf{x})$ is the set of points $\mathbf{x} \in \mathscr{X}$ where $f(\mathbf{x})$ has a constant value.It is basically a cross section of $y = f(\mathbf{x})$ where it takes a constant value, say $y = c$. For different constants c, we obtain different level sets, which form the so-called *contour plot*.

- **Parametric Functions.** A parametric function is a map parametrized by some parameters $\mathbf{w}$, belonging to a set $\mathscr{W}$. We write $f(\mathbf{x};\mathbf{w})$ or $f_{\mathbf{w}}(\mathbf{x})$ to indicate that the function f has input $\mathbf{x}$ and parameter w. Functions that differ only in the specification (instantiation) of the parameter w belong to the same *function class* $\mathscr{C}$ or *hypothesis space* $\mathscr{H}$.

  Typical examples of function classes include:

  - **Linear functions**: $f(\mathbf{x};\mathbf{w},b) = \mathbf{w}^\top\mathbf{x} + b$, where $(\mathbf{w},b)$ are the parameters
  - **Polynomial functions**: $f(x,\mathbf{w}) = \sum_{k=0}^{K} w_k x^k$, where $\mathbf{w} = \{w_0,\ldots,w_K\}$ are the coefficients of the polynomial and K is the order
  - **Logistic Functions:** $f(\mathbf{x};\mathbf{w}) = (1 + e^{-\mathbf{w}^\top\mathbf{x}})^{-1}$, which returns a number between zero and one

- **Derivatives.** The derivative of a real-valued function $f : \mathbb{R} \to \mathbb{R}$ is denoted by $f'(x)$ and it is defined as:

$$f'(x) = \frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}. \tag{60}$$

  Informally, it is a function representing the the slope of the tangent line to the point $(x, f(x))$. Functions whose derivative exists at each point in their domain are said differentiable.

- **Gradient.** For a multivariable real-valued function $f : \mathbb{R}^N \to \mathbb{R}$, the gradient is:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_N} \end{bmatrix} \tag{61}$$

  i.e., it is the vector that stacks the partial derivatives $\nabla f(\mathbf{x})_i = \frac{\partial f(\mathbf{x})}{\partial x_i}$ for $i = 1,\ldots,N$.

  - **Differentiable function.** A function $f : \mathbb{R}^N \to \mathbb{R}$ is differentiable if $\mathbf{dom} f$ is open and the gradient $\nabla f(\mathbf{x})$ exists at each $x \in \mathbf{dom} f$.

- **Hessian.** For a multivariable real-valued function $f : \mathbb{R}^N \to \mathbb{R}$, the Hessian is:

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_N} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & & \frac{\partial^2 f}{\partial x_2 \partial x_N} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_N \partial x_1} & \frac{\partial^2 f}{\partial x_N \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_N^2} \end{bmatrix} \tag{62}$$

i.e., it is the matrix that stacks all the second partial derivatives $(\mathbf{H}f)_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

- **Twice differentiable.** A function $f : \mathbb{R}^N \to \mathbb{R}$ is twice differentiable if $\mathbf{dom} f$ is open and the Hessian $\nabla^2 f(\mathbf{x}) \in \mathbf{S}^N$ exists at each $\mathbf{x} \in \mathbf{dom} f$.

- **Jacobian.** For a multivariable vector-valued function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^M$, the Jacobian is:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \cdots & \frac{\partial \mathbf{f}}{\partial x_N} \end{bmatrix} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \frac{\partial f_M}{\partial x_N} \end{bmatrix}. \tag{63}$$

i.e., it is the matrix that stacks all the first partial derivatives of $\mathbf{f}$. Notice that the Hessian of a real-valued function is the Jacobian of its gradient.

.6. Convex Analysis

- **Line.** The line between two points $\mathbf{x}_1$ and $\mathbf{x}_2$ is the set of points or segment $\{\mathbf{x} \,|\mathbf{x} = \theta\mathbf{x}_1 + (1-\theta)\mathbf{x}_2,\ \theta \in [0,1]\}$.

- **Convex set.** A set $\mathscr{C}$ is convex if, for every two points $\mathbf{x}_1, \mathbf{x}_2 \in \mathscr{C}$ and any scalar $0 \le \theta \le 1$, the line joining $\mathbf{x}_1$ and $\mathbf{x}_2$ belongs to the set, i.e., $\theta\mathbf{x}_1 + (1-\theta)\mathbf{x}_2 \in \mathscr{C}$. Fundamental convex sets include:
  - **hyperplane**: it is a set of the form $\{\mathbf{x}|\mathbf{w}^\top\mathbf{x} = b\}$, with $\mathbf{w} \neq \mathbf{0} \in \mathbb{R}^N$ and $b \in \mathbb{R}$. For instance, in a 2D space, it is a line, while in a 3D space it is a 2D plane. An example of hyperplane is shown in Figure .
  - **halfspace**: it is a set of the form $\{\mathbf{x}|\mathbf{w}^\top\mathbf{x} \le b\}$, with $\mathbf{w} \neq \mathbf{0}$. An hyperplane divides the space in two halfspaces. An example of halfspace is shown in Figure .
  - **polyhedra**: it is a set of the form $\{\mathbf{x}|\mathbf{w}_i^\top\mathbf{x} \preceq b_i \cap \mathbf{w}_j^\top\mathbf{x} = b_j, \text{for some } i, j\}$. In other words, it is the intersection of a finite number of hyperplanes and halfspaces. An example of polyhedra is shown in Figure .
  - **positive semidefinite cone**: it is the set of positive semidefinite matrices (i.e. the set of matrices with all the eigenvalues greater than or equal to zero). It is denoted as $\mathbb{S}_+^n = \left\{\mathbf{X} \in \mathbb{S}^N \,|\, \mathbf{X} \succcurlyeq 0\right\}$.

- **Non convex set.** A non convex set is a set not satisfying the the definition of convexity. As an example of a non convex set consider $\mathscr{X} = \{[1,2] \cup [3,4]\}$. This is because the line joining $x_1 = 1.5$ and $x_2 = 3.5$ is not contained in $\mathscr{X}$.

  In Figure 9 are shown examples of convex and non convex sets.

Figure 9: Examples of convex and non-convex sets. The three sets on the upper part of the figure are convex, since every line segment between any two points in the set belongs to the set. Differently, the three sets on the bottom part of the figure are not convex, since there exist points in the set such that the line joining them (dotted in the figure) does not belong to the set.

- **Convex Functions.** A function $f(\cdot)$ is *convex* if $\mathbf{dom} f$ is a convex set and if :

$$f(\theta\mathbf{x}_1 + (1-\theta)\mathbf{x}_2) \le \theta f(\mathbf{x}_1) + (1-\theta) f(\mathbf{x}_2), \text{ for all } \mathbf{x}_1, \mathbf{x}_2 \in \mathbf{dom} f, \text{ for all } \theta \in [0,1] \tag{64}$$

This condition is known also as **zero-th order** convexity condition. Intuitively, a function $f(\cdot)$ is convex when, taking any two points of its domain, the line connecting them lies above the graph of $f(\cdot)$. We say that $f(\cdot)$ is *concave* if $-f(\cdot)$ is convex. Because for an affine function the inequality in (64) holds with equality, affine functions (and hence linear ones) are either convex and concave. From a practical point of view, we study convex functions because they have a unique minimal value (see Figure 10). We shall detail next how to find the latter.

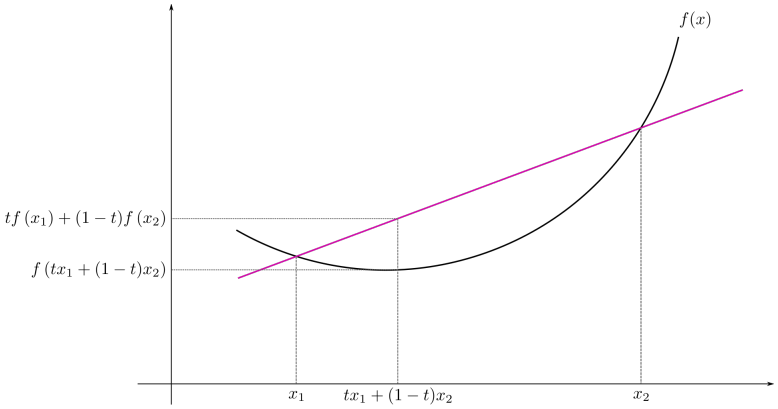


Figure 10: Convex function. The purple line connecting the points (x_1, y_1) and (x_2, y_2) lies above the graph of $f(\cdot)$, when this is evaluated at points in the interval $[x_1, x_2]$. It has a minimal value y_0, associated to the point x_0. From Wikipedia.

.6.1. Convexity Conditions

- **First order condition.** A differentiable function $f(\cdot)$ is convex if and only if:

$$f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) \text{ for } \mathbf{x}, \mathbf{y} \in \mathbf{dom} f \tag{65}$$

- **Second order condition.** A twice differentiable function $f(\cdot)$ is convex if and only if:

$$\nabla^2 f(\mathbf{x}) \succcurlyeq 0 \text{ for all } \mathbf{x} \in \mathbf{dom} f \tag{66}$$

  This condition can be interpreted geometrically as the requirement for the graph of the function to have a positive curvature.

.7. Optimization Problems

This section presents first the basic definitions of optimization theory and then illustrates them with examples.

.7.1. Basic Definitions

In its more general form, an optimization problem reads as:

$$\begin{array}{ll} \min_{\mathbf{x}} & f_0(\mathbf{x}) \\ \text{subject to} & f_i(\mathbf{x}) \leq 0, \quad i = 1, \ldots, m \\ & h_i(\mathbf{x}) = 0, \quad i = 1, \ldots, p \end{array} \tag{67}$$

where:

- $\mathbf{x} \in \mathbb{R}^N$ is the optimization variable
- $f_0 : \mathbb{R}^N \to \mathbb{R}$ is the cost or objective function
- $f_i : \mathbb{R}^N \to \mathbb{R}, i = 1, \ldots, m$ are the inequality constraint functions
- $h_i : \mathbb{R}^N \to \mathbb{R}, i = 1, \ldots, p$ are the equality constraint functions
- **Problem domain.** The *domain* $\mathscr{D}$ of the optimization problem is the set of points for which the objective function and the constraint functions are defined, i.e., $\mathscr{D} = \bigcap_{i=0}^{m} \mathbf{dom} f_i \cap \bigcap_{i=1}^{p} \mathbf{dom}\, h_i$.
- **Feasible point.** A point $\mathbf{x}_j \in \mathscr{D}$ is said to be *feasible* if it satisfies all the constraints in the problem, i.e. $f_i(\mathbf{x}_j) \leq 0$ for $i = 1, \ldots, m$ and $h_i(\mathbf{x}_j) = 0$ for $i = 1, \ldots, p$.
- **Feasible set.** The set of all points that satisfy the constraints is called feasible set or constraint set.
- **Infeasible problem.** A problem is infeasible if there are not points satisfying the constraints.

- **Unconstrained problem.** An optimization problem is unconstrained if it has the expression:

$$\min_{\mathbf{x}} f_0(\mathbf{x}) \tag{68}$$

  i.e., when there are no constraints ($m = p = 0$ in (67)). In this introductory chapter, we will mainly recap unconstrained problems and the algorithms to solve them, while direct interested readers in the constrained case to [Boyd].

- **Convex problem.** A problem is convex when functions $f_0, f_1, \dots, f_m$ are convex and the equality constraints $h_1, \dots, h_p$ are affine, i.e. they are of the form $h_i(\mathbf{x}) = \mathbf{a}_i^\top \mathbf{x}$ for some vectors $\mathbf{a}_1, \dots, \mathbf{a}_p$. Stacking together row-wise vectors $\mathbf{a}_i$, we often represent the affine constraints in the matrix-vector form $\mathbf{Ax} = \mathbf{b}$.

In the sequel we provide two examples of optimization problems.

Linear regression. Suppose we have a dataset $\mathcal{T} = \{\mathbf{x}_i, y_i\}_{i=1}^m$ composed of input-output pairs $\mathbf{x}_i \in \mathbb{R}^N$ and $y_i \in \mathbb{R}$. These may be for instance characteristics of a house, such as number of rooms, squared meters, construction year, etc., encoded in the vector $\mathbf{x}_i$, and the price of the house, encoded in the scalar y_i. The input $\mathbf{x}_i$ is referred to as the *feature vector*, while the output y_i is usually called *target* or *label*. In linear regression analysis the goal is to find the best linear function $\hat{f}$ that maps each input $\mathbf{x}_i$ into the associated output y_i as accurately as possible in a least squares sense. In other words, $\hat{f}$ is a parametric linear function with parameter $\mathbf{w} \in \mathbb{R}^N$, i.e. $\hat{f}(\mathbf{x}; \mathbf{w}) = \mathbf{w}^\top \mathbf{x}$, and linear regression approaches the problem of finding the parameter $\mathbf{w}$ by minimizing the sum of squared residuals $r_i = y_i - \mathbf{w}^\top \mathbf{x}_i$. That is,:

$$\min_{\mathbf{w}} \sum_{i=1}^{m} (y_i - \mathbf{w}^\top \mathbf{x}_i)^2 = \min_{\mathbf{w}} \ \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2, \tag{69}$$

where $\mathbf{X} = [\mathbf{x}_1^\top, \dots, \mathbf{x}_m^\top]^\top$ is the feature matrix containing in the ith row the feature vector $\mathbf{x}_i$ and $\mathbf{y} = [y_1, \dots, y_m]^\top$ is the target vector.

Ridge Regression. Consider now that we want to find a vector $\mathbf{w}$ for problem (69) which has a bounded norm. These may be for instance voltages we can induce in different parts of a circuit and we have a maximum capacity of the total amount we can induce. We can modify problem (69) as:

$$\begin{aligned} &\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 \\ &\text{s.t.} \quad \|\mathbf{w}\|^2 \leq \omega \end{aligned} \tag{70}$$

We can transform problem (72) as an unconstrained problem as[1]:

$$\min_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda \|\mathbf{w}\|^2, \tag{71}$$

for a particular choice of $\lambda \geq 0$. This problem is called *ridge regression*, which goal is to minimize a weighted combination of two terms: the sum of squared residuals and the squared norm of the vector parameter $\mathbf{w}$. The (hyper-)parameter λ trades-off these two terms.

Concave Problems. Sometimes function $f_0(\cdot)$ may be a concave function. In these instances, it leads to a maximization problem, i.e.,:

$$\max_{\mathbf{x}} f_0(\mathbf{x}) \tag{72}$$

However, since $-f_0(\cdot)$ is a convex function, we can rephrase problem (72) into a minimization problem as in (68) by minimizing $-f_0(\mathbf{x})$. However, you may also encounter optimization problems involving functions that are neither convex nor concave. In that case, tools from non convex optimization literature are needed, but we will not discuss them here. More information in this regard can be found in [2].

.7.2. Optimality

Our goal is to minimize the cost function $f_0(\cdot)$. This is tantamount to finding the *minimum* value of f_0, denoted with $p^\star$. In addition, we want to find also the argument (*minimizer*) achieving it, that we denote with $\mathbf{x}^\star$.

[1] This is done with the use of the Laplacian function.

- **Minimum.** The smallest value $p^\star$ for which a function $f_0(\cdot)$ is defined, is called minimum of the function. For unconstrained problem, this value coincide also with the minimum of the problem. Moreover, if the function $f_0(\cdot)$ is convex, the minimum $p^\star$ is *unique*.

- **Minimizer.** Given a function $f_0(\cdot)$, the argument $\mathbf{x}^\star$ of the function that attains the minimum, i.e. such that $f_0(\mathbf{x}^\star) = p^\star$, is called minimizer of the function. For a convex function, there may exists multiple points $\mathbf{x}_i^\star$ attaining the minimum. In this case we have a set of minimizers, that we denote as $\mathscr{X}_{opt}$. In Figure **??** are shown the multiple minimizers of function.

- **Infeasible.** If a problem is infeasible (cfr. Basic Definitions), then $p^\star = +\infty$.

- **Unbounded.** A problem is unbounded below if does not exist a finite lower-bound of $f_0(\cdot)$. In this case then $p^\star = -\infty$.

- **Globally optimal point.** A feasible point $\mathbf{x}$ is (globally) **optimal** if $f_0(\mathbf{x}) = p^\star$, i.e. it is a minimizer of the function.

- **Locally optimal point.** A feasible point $\mathbf{x}$ is **locally optimal** if it minimizes the function in a region of the feasible set.

For convex optimization problems, any locally optimal point is also globally optimum. This is not the case for non convex functions, which admit multiple minimum points, that are local minima.

Optimality Conditions

For an unconstrained convex problem, the following are necessary and sufficient condition for a point $\mathbf{x}^\star$ to be optimal for a function $f_0(\mathbf{x})$:

- **Zeroth order.** For all $\mathbf{x} \in \mathbf{dom} f$:
$$f_0(\mathbf{x}^\star) \leq f_0(\mathbf{x}) \tag{73}$$

- **First order.**
$$\nabla f(\mathbf{x}^\star) = \mathbf{0} \tag{74}$$

- **Second order.**
$$\nabla^2 f(\mathbf{x}^\star) \succ 0 \tag{75}$$

In other words, the zeroth order optimality condition requires that the function value at $\mathbf{x}^\star$ is smaller than or equal to the function value of any other point (difficult to prove). The first order optimality condition requires that the gradient vanishes in the optimal point $\mathbf{x}^\star$ (often easy to prove). This also enable us to find $\mathbf{x}^\star$ by solving the system of equation (74). The second order optimality condition requires the Hessian of the function to be strictly positive definite when evaluated at the point $\mathbf{x}^\star$.

.8. Typical Optimization Problems

In this section we describe commons optimization problems and their respective solutions.

- **Linear Program.** A linear program is a problem in which the objective and constraint functions are affine (hence convex) of the form:
$$\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & \mathbf{c}^\top \mathbf{x} \\ \text{subject to} & \mathbf{G}\mathbf{x} \preceq \mathbf{h} \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \end{array} \tag{76}$$
for some constant vector $\mathbf{c}$ and matrices $\mathbf{A}, \mathbf{G}$. The constraints define a convex set to search the solution, in particular a polyhedron. The optimal point, if exists[2], always lies in one of its vertices, as shown in Figure 11a. Problem (76) without any constraints is unbounded below, i.e. $p^\star = -\infty$.

[2] If the constraints are inconsistent or if the polyhedron is unbounded in the direction of the gradient, a solution does not exist.

- **Quadratic Program.** A quadratic program is an optimization problem of the form:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & \frac{1}{2}\mathbf{x}^\top \mathbf{P}\mathbf{x} + \mathbf{c}^\top \mathbf{x} + d \\ \text{subject to} \quad & \mathbf{G}\mathbf{x} \preceq \mathbf{h} \\ & \mathbf{A}\mathbf{x} = \mathbf{b} \end{aligned} \tag{77}$$

where $\mathbf{P} \in \mathbb{S}_+^N$, $\mathbf{G} \in \mathbb{R}^{m \times N}$ and $\mathbf{A} \in \mathbb{R}^{p \times N}$. It is a convex optimization problem with quadratic objective function and affine constraints. Also in this case, the feasible set is a polyhedron, and we show the overall picture of a quadratic program in Figure 11b.

(a) (b)

Figure 11: (a) Feasible set ($\mathscr{P}$) and level curves for a linear program. The optimal point $\mathbf{x}^\star$ lies in one of the vertex of $\mathscr{P}$, which is a polyhedron; and (b) Feasible set ($\mathscr{P}$) and level curves for a quadratic program. The feasible set is a polyhedron, while the level curves are elliptical. Figure from [1].

Problem (77) without any constraints has the closed form solution $\mathbf{x}^\star = -\mathbf{P}^\dagger \mathbf{c}$, , where $\mathbf{P}^\dagger$ denotes the Moore-Penrose pseudo-inverse of $\mathbf{P}$. This solution is obtained for instance by applying the first-order optimality condition. Note that a linear program is a special case of quadratic problems, by taking $\mathbf{P} = \mathbf{0}$.

Example: Least Squares. Consider the linear regression problem (69). This formulation, as mentioned earlier, is called *least-squares*, and our goal is to minimize the mean squared error between our linear predictions $\hat{y}_i = \mathbf{w}^\top \mathbf{x}_i$ and the true observation y_i. In vector form:

$$\mathbf{w}^\star = \quad \underset{\mathbf{w}}{\text{argmin}} \{ f_0(\mathbf{w}) := \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 \}. \tag{78}$$

where we make now explicit that we are interested in the argument of the function through the argmin notation. This is a quadratic problem in $\mathbf{w}$. Indeed expanding the objective function we have:

$$\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) = \mathbf{y}^\top \mathbf{y} - 2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} \tag{79}$$

Because matrix $\mathbf{X}^\top \mathbf{X} \in \mathbb{S}_+^N$, the optimization problem is convex. To find the solution, we solve $\nabla f_0(\mathbf{w}) = \mathbf{0}$ and obtain:

$$\mathbf{X}^\top \mathbf{X}\mathbf{w} = \mathbf{X}^\top \mathbf{y} \tag{80}$$

from which we obtain in closed form the optimal solution:

$$\mathbf{w}^\star = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y} = \mathbf{X}^\dagger \mathbf{y} \tag{81}$$

.8.1. Regularized Problems

A regularized optimization problem is the one with the form:

$$\min_{\mathbf{x}} \quad f(\mathbf{x}) + \lambda g(\mathbf{x}) \tag{82}$$

The goal of a regularized problem is to make both $f_0(\mathbf{x})$ and $g(\mathbf{x})$ small. Scalar $\lambda \geq 0$ is a trade-off parameter between purely $f(\mathbf{x})$ and $g(\mathbf{x})$. For $\lambda \to 0$ we minimize mainly $f(\mathbf{x})$, while for $\lambda \to \infty$ we mainly minimize $g(\mathbf{x})$. For this, many times, we say that $g(\mathbf{x})$ represents a *penalty* term. If both $f(\mathbf{x})$ and $g(\mathbf{x})$ are convex, then the overall optimization problem is convex, due to the property that the non-negative sum of convex functions is

convex. Problem (71) is an example of regularized problem. There $f(\mathbf{x}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ and $g(\mathbf{x}) = \|\mathbf{w}\|^2$, where we imposed as penalty that the solution has also a small norm.

In the sequel we focus on three different regularization terms, namely, the ℓ_0, ℓ_1 and ℓ_2 norms for the optimization variables.

- ℓ_0 **norm.** The function $g(\mathbf{x}) = \|\mathbf{x}\|_0 = \#\{i : x_i \neq 0\}$ is called ℓ_0 pseudo-norm. The regularized optimization problem is then:

$$\min_{\mathbf{x}} \quad f(\mathbf{x}) + \lambda \|\mathbf{x}\|_0 \tag{83}$$

  Because the ℓ_0 norm counts the number of non-zero entries in the vector $\mathbf{x}$, this regularization term enforces a sparse solution for the problem, i.e. a solution with many entries equal to zero. This is for instance the case of a sensor network, in which the number of switched on sensors corresponds to higher maintenance and power costs. Because ℓ_0 is non convex and non differentiable, it is a term difficult to handle. The best convex approximation of the ℓ_0 norm is the ℓ_1 norm, explained next.

- ℓ_1 **norm.** The function $g(\mathbf{x}) = \|\mathbf{x}\|_1 = \sum_{i=1}^{N} |x_i|$ is called ℓ_1 norm. The regularized optimization problem is:

$$\min_{\mathbf{x}} \quad f(\mathbf{x}) + \lambda \|\mathbf{x}\|_1 \tag{84}$$

  Because the ℓ_1 norm is the closest *convex relaxation* of the ℓ_0 norm, this regularization term is often used as a heuristic for finding a sparse solution. It is convex but not differentiable, leading to non-smooth optimization problems.

- ℓ_2 **norm (squared).** The function $g(\mathbf{x}) = \|\mathbf{x}\|_2^2 = \sum_{i=1}^{N} x_i^2$ is called squared ℓ_2 norm. The regularized optimization problem is:

$$\min_{\mathbf{x}} \quad f(\mathbf{x}) + \lambda \|\mathbf{x}\|_2^2 \tag{85}$$

  Because the ℓ_2 norm measures the length of the vector $\mathbf{x}$, this regularization term favours solutions close to $\mathbf{0}$. It is squared because it eliminates the square root and it is easier to handle mathematically. Because it is convex, the overall optimization problem is convex, as long as $f(\mathbf{x})$ is convex.

  Example: Tikhonov regularization. The Ridge Regression problem (71) is also called *Tikhonov regularization*, and it is one of the most common form of regularized problems. It is a convex quadratic optimization problem:

$$\mathbf{w} = \quad \operatorname*{argmin}_{\mathbf{w}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda \|\mathbf{w}\|^2. \tag{86}$$

  Being an unconstrained problem with differentiable objective function, we can find analytically its solution. Indeed, the function can be rewritten as:

$$f(\mathbf{w}) = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda \mathbf{w}^\top \mathbf{w} = \mathbf{y}^\top \mathbf{y} - 2\mathbf{y}^\top \mathbf{X}\mathbf{w} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X}\mathbf{w} + \lambda \mathbf{w}^\top \mathbf{w}$$

  By setting $\nabla_{\mathbf{w}} f(\mathbf{w}) = \mathbf{0}$ we obtain $\nabla_{\mathbf{w}} f(\mathbf{w}) = -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X}\mathbf{w} + 2\lambda \mathbf{w} = \mathbf{0}$, from which we find the optimal solution:

$$\mathbf{w}^\star = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y} \tag{87}$$

  It resembles the already seen form of least squares solution (81), yet this time with a very fundamental difference. Differently from (81), here we are not involving a pseudoinverse: the regularization term "repairs" the (possible) rank-deficiency of the matrix $\mathbf{X}$. Indeed $\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I} > 0$ for any $\lambda > 0$.

.9. Summary and Further Readings

In this chapter, we introduced the notion of convexity for sets and functions, together with illustrative examples. As pointed out, (convex) optimization is a fundamental framework for each learning algorithm, since the latter requires to solve an optimization problem. If the problem is convex, whose necessary and sufficient conditions have been outlined in Section .7.2, then there is a global minimum solving the problem. Although we have seen in Section .8 that it is possible in some cases to find the optimal point (attaining the minimum) in closed form, this is not always the case. In the next section, we show how to solve optimization problems through iterative algorithms, i.e., methods that generate a sequence of points approaching the optimal solution of the problem. For an in-depth study on the subject, we recommend the following references.

Bibliography

[1] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[2] Prateek Jain and Purushottam Kar. "Non-convex optimization for machine learning". In: *arXiv preprint arXiv:1712.07897* (2017).

Appendix C: Iterative Algorithms

In the previous chapter, we defined convex optimization problems. Now we show how to solve these problems via iterative methods, i.e. methods which find a "good" solution for the problem in an iterative way, refining the estimate solution at each iteration.

.9.1. Descent Methods

Suppose we want to minimize a convex and differentiable function $f : \mathbb{R}^N \to \mathbb{R}$. A point $\mathbf{x}^\star$ is the global minimum of $f(\cdot)$ if and only if $\nabla f(\mathbf{x}^\star) = \mathbf{0}$. In a parabola, this is represented by the point associated to the vertex of the curve. The intuition behind *descent methods* can be given with the following physical analogy: by placing a ball in a point of the parabola with high potential energy, the ball will start moving downhill reaching a point with low potential energy (possibly the one with the minimum energy).

More formally, a descent method computes a sequence of points $\mathbf{x}^0, \mathbf{x}^1, \ldots$, such that successive updates yield lower function values, i.e., $f(\mathbf{x}^{k+1}) \leq f(\mathbf{x}^k)$ for increasing iteration number k. In particular, $f(\mathbf{x}^k) \to f(\mathbf{x}^\star)$ as $k \to \infty$. The sequence of points $\mathbf{x}_0, \mathbf{x}_1, \ldots$, is called the *minimizing sequence*.

Minimizing sequence and descent direction. The minimizing sequence is produced by following the recursive equation:

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \Delta\mathbf{x}^k, \quad k = 1, 2, \ldots \tag{88}$$

where $\Delta\mathbf{x}^k \in \mathbb{R}^N$ is called *search direction* and $\alpha^k \in (0, 1)$ is the *step-size* (or learning rate). The superscript k indicates that these quantities (may) change at every iteration. Among all the possible search directions, we have to guarantee that $\Delta\mathbf{x}^k$ is a descent direction, i.e. a direction for which holds $f(\mathbf{x}^{k+1}) \leq f(\mathbf{x}^k)$.

From the first-order convexity condition, we know that if f is convex, then $f(\mathbf{x}^{k+1}) \geq f(\mathbf{x}^k) + \nabla f(\mathbf{x}^k)^\top (\mathbf{x}^{k+1} - \mathbf{x}^k)$ for all $\mathbf{x}^{k+1}$; thus a descent direction in a descent method must satisfy:

$$\nabla f(\mathbf{x}^k)^\top (\mathbf{x}^{k+1} - \mathbf{x}^k) \leq 0, \tag{89}$$

meaning that the search direction $\Delta\mathbf{x}^k := (\mathbf{x}^{k+1} - \mathbf{x}^k)$ should be negatively aligned with the gradient $\nabla f(\mathbf{x}^k)$.

As we will see, the step-size α has a big impact on the convergence of the algorithm, since it determines where, along the line $\mathbf{x}^k + \alpha\Delta\mathbf{x}^k$, the next iterate $\mathbf{x}^{k+1}$ will be. If α is too small the algorithm will be slow, while an α too big may even not lead to convergence near the optimal point.

There are typically two approaches to halt a descent method. First we fix a small scalar $\epsilon > 0$ and halt the algorithm when $\|\nabla f(\mathbf{x}^k)\| \leq \epsilon$. If $\epsilon \to 0$, this condition ensures that the gradient evaluated at iteration k is near zero so we are close to the optimum. Another condition is to fix a number K of iterations and halt the algorithm when such iterations are completed. The pseudocode of a general descent method is reported in Algorithm 3.

Algorithm 3 General descent method

Require: initial starting point $\mathbf{x}^{(0)}$, stopping criterion ϵ

$k \leftarrow 0$
while not converged **do**
  Determine $\Delta\mathbf{x}^k$ // search direction
  Choose $\alpha^k > 0$ // step-size
  $\mathbf{x}^{k+1} := \mathbf{x}^k + \alpha^k \Delta\mathbf{x}^k$ // Update variable
  Check convergence (ϵ)
  $k \leftarrow k + 1$
end while
return $\mathbf{x}^{k+1}$

Gradient Descent. It is a descent algorithm with update $\Delta\mathbf{x}^k = -\nabla f(\mathbf{x}^k)$, i.e., the descent direction is the negative gradient of $f(\cdot)$ evaluated at $\mathbf{x}^k$. It is a descent direction because condition (89) holds: indeed, $-\nabla f(\mathbf{x}^k)^\top \nabla f(\mathbf{x}^k) = -\|\nabla f(\mathbf{x}^k)\|^2 \leq 0$. The pseudocode of gradient descent, which is just the specialization of Algorithm 3 with $\Delta\mathbf{x}^k = -\nabla f(\mathbf{x}^k)$, is reported in Algorithm 4.

Algorithm 4 Gradient descent method

Require: initial starting point $\mathbf{x}^{(0)}$, stopping criterion ϵ
1: $k \leftarrow 0$
2: **while** not converged **do**
3: $\Delta\mathbf{x}^k := -\nabla f(\mathbf{x}^k)$
4: Choose $\alpha^k > 0$ // step-size
5: $\mathbf{x}^{k+1} := \mathbf{x}^k - \alpha^k \nabla f(\mathbf{x}^k)$ // Update variable
6: Check convergence (ϵ)
7: $k \leftarrow k+1$
8: **end while**
9: **return** $\mathbf{x}^{k+1}$

Steepest Descent. Gradient descent ensures we update our algorithm along the *direction of steepest descent* of function $f(\cdot)$ at the point $\mathbf{x}^k$; i.e., $-\nabla f(\mathbf{x}^k)$. This means that our next iterate $\mathbf{x}^{k+1}$ should move along the line $\mathbf{x}^k - \alpha^k \nabla f(\mathbf{x}^k)$. Which α to pick? A possible answer is given by the steepest descent method: the step-size is chosen in such a way to yield the highest decrease in the cost function along the negative gradient, i.e.,:

$$\alpha^{k\star} = \underset{\alpha}{\operatorname{argmin}} f(\mathbf{x}^k - \alpha \nabla f(\mathbf{x}^k)). \tag{90}$$

That is, for each iteration k, we solve an optimization problem in the variable α that minimizes the function along the line $\mathbf{x}^k - \alpha^k \nabla f(\mathbf{x}^k)$. Many times, this optimization problem is approximated (grid-search): a finite number of points for α in an interval $[\alpha_{\max}, \alpha_{\max}]$ is taken and the function is evaluated in all. Then the value of α yielding the lowest function value is chosen. In order to avoid the computation of $\alpha^{k\star}$ for every iteration k, in practice it is sufficient to fix a constant value of α, which guarantees convergence, all over the descent method.

.9.2. Solving Linear Systems of Equation with Iterative Methods

Iterative methods can also be used to solve linear system of equations (LSE), i.e., systems of the form $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{x}$ is an unknown vector to be determined. We assume that the system is solvable, meaning that it has the unique solution $\mathbf{x}^\star = \mathbf{A}^{-1}\mathbf{b}$.

Gradient descent. Recognizing the optimal point $\mathbf{x}^\star$ to be also the global optimum of the function $f(\mathbf{x}) = \frac{1}{2}\|\mathbf{b} - \mathbf{Ax}\|_2^2$, solving the LSE means solving the optimization problem:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \ \frac{1}{2}\|\mathbf{b} - \mathbf{Ax}\|_2^2$$

If the minimum $\mathbf{x}^\star$ of this function is zero, we have solved our problem exactly; otherwise, we have approximated the original solution. We want to approach the minimum of the function, i.e., to solve the LSE, with gradient descent.

The gradient of $f(\cdot)$ is given by $\nabla f(\mathbf{x}) = \mathbf{A}^\top(\mathbf{Ax} - \mathbf{b})$. Provided an initial starting point $\mathbf{x}^0$ at random, repeat:

$$\begin{aligned} \mathbf{x}^{k+1} &= \mathbf{x}^k - \alpha \nabla f(\mathbf{x}^k) \\ &= \mathbf{x}^k - \alpha \mathbf{A}^\top(\mathbf{Ax}^k - \mathbf{b}) \end{aligned} \tag{91}$$

until stopping criterion is satisfied. The exact solution will be achieved when the error term $\mathbf{e}^k = \mathbf{Ax}^k - \mathbf{b}$ equals the zero vector.

Jacobi method. Let $\mathbf{A} = \mathbf{D} + \mathbf{R}$, where $\mathbf{D}$ is the diagonal matrix containing the diagonal elements of $\mathbf{A}$ (which we assume does not contain zero elements in its diagonal) and $\mathbf{R}$ the off-diagonal elements. Solving $\mathbf{Ax} = \mathbf{b}$ means then to solve $\mathbf{Dx} + \mathbf{Rx} = \mathbf{b}$, which in turns leads to $\mathbf{x} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{Rx})$. This suggests the idea to solve $\mathbf{Ax} = \mathbf{b}$ iteratively with the recursion:

$$\mathbf{x}^{k+1} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{Rx}^k) \tag{92}$$

In other words, the Jacobi method solves at each iteration the following system of equations: $\mathbf{Dx}^{k+1} + \mathbf{Rx}^k = \mathbf{b}$. It enjoys a parallel implementation.

Conjugate gradient. When matrix $\mathbf{A}$ is positive-definite, we can use the conjugate method to solve the LSE. This method exploits the fact that solving the LSE is equivalent to solve the quadratic optimization problem:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \quad f(\mathbf{x}) := \frac{1}{2}\mathbf{x}^\top \mathbf{Ax} - \mathbf{b}^\top \mathbf{x} \tag{93}$$

Since matrix $\mathbf{A}$ is positive definite, problem (93) is convex and the unique global minimum occurs at the point $\mathbf{x}$ for which the gradient is equal to zero, i.e., $\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b} = \mathbf{0}$. The other particularity of such a problem is that the LSE residual $r(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$ coincides with the gradient of the function, i.e., $r(\mathbf{x}) = \nabla f(\mathbf{x})$. The conjugate gradient method approaches the minimization of $f(\mathbf{x})$ by exploiting these facts, whose steps for solving problem (93) are provided in Algorithm 5 (see [5] for details).

Algorithm 5 Conjugate Gradient

Require: $\mathbf{x}^{(0)}$, linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\epsilon > 0$
1: $\mathbf{r}^0 = \mathbf{A}\mathbf{x}^0 - \mathbf{b}$
2: $\mathbf{d}_0 = \mathbf{r}^0$
3: $k \leftarrow 0$
4: **while** not converged **do**
5: $\quad \alpha^k := \frac{\mathbf{r}^{k\top}\mathbf{r}_k}{\mathbf{d}_k^\top \mathbf{A}\mathbf{d}_k}$
6: $\quad \mathbf{x}^{k+1} := \mathbf{x}^k + \alpha^k \mathbf{d}_k$
7: $\quad \mathbf{r}^{k+1} := \mathbf{r}^k + \alpha^k \mathbf{A}\mathbf{d}_k$
8: $\quad \beta^{k+1} := \frac{\mathbf{r}^{(k+1)\top}\mathbf{r}^{k+1}}{\mathbf{r}^{k\top}\mathbf{r}^k}$
9: $\quad \mathbf{d}_{k+1} := \mathbf{r}^{k+1} + \beta^{k+1}\mathbf{d}_k$
10: $\quad$ Check convergence (ϵ)
11: $\quad k \leftarrow k+1$
12: **end while**
13: **return** $\mathbf{x}^{k+1}$

The recursive update formula for iteration $k+1$ (line 6) is given by:

$$\mathbf{x}^{k+1} := \mathbf{x}^k + \alpha^k \mathbf{d}_k, \tag{94}$$

where the step-size α^k is set as $\alpha^k = \frac{\mathbf{r}^{k\top}\mathbf{r}_k}{\mathbf{d}_k^\top \mathbf{A}\mathbf{d}_k}$, which is proven to be the optimal value for iteration k. Here $\mathbf{r}_k$ is the LSE residual vector, while $\mathbf{d}_k$ is the search direction of iteration k. The peculiarity of this algorithm is that all the search directions are mutually conjugate, one with each other, with respect to $\mathbf{A}$, i.e., $\mathbf{d}_j \mathbf{A}\mathbf{d}_k = 0$ for all j, k. This can be achieved by updating the search direction at each iteration as $\mathbf{d}_k = \mathbf{r}^k + \beta^k \mathbf{d}_{k-1}$, where $\mathbf{r}^k = \mathbf{r}^{k-1} + \alpha^{k-1}\mathbf{A}\mathbf{d}_{k-1}$ is the update of the residual vector, and the parameter β^k is determined by satisfying the requirement that $\mathbf{d}_k$ and $\mathbf{d}_{k-1}$ are mutually conjugate w.r.t. $\mathbf{A}$. After some calculations, it can be proved that the optimal β^k is $\beta^k = \frac{\mathbf{r}^{(k+1)\top}\mathbf{r}^{k+1}}{\mathbf{r}^{k\top}\mathbf{r}^k}$.

Remark. Gradient descent is the simplest, the lightest and the most intuitive between the methods introduced to solve LSEs. However, it suffers in terms of convergence speed, i.e. it has low convergence rate. Differently, conjugate gradient is faster, but it requires more assumptions and a more complex implementation. Jacobi method is an intermediate solution, which also enjoys a parallel implementation.

.10. Data-Driven Optimization via Iterative Algorithms

We now introduce a class of algorithms that concern solving optimization problems in a data-driven fashion. These algorithms do not require a priori the entire sequence of data but rather operate on-the-fly.

.10.1. Least Mean Squares

Consider: *i)* a *data* matrix $\mathbf{X} \in \mathbb{R}^{m \times N}$, with m denoting the number of observations/samples and N denoting the number of features of each sample; *ii)* a *target* vector $\mathbf{y} \in \mathbb{R}^m$, where each entry y_i is a *label* associated to one observation; *iii)* a *weight* vector $\mathbf{w} \in \mathbb{R}^N$, that is unknown. A simple yet effective way to find the best weight vector $\mathbf{w}^\star$ that relates in a linear fashion each label y_i to the features of observation i, i.e. $[X_{i1}, \ldots, X_{iN}]^\top$, is given by the least squares formulation:

$$\underset{\mathbf{w}}{\operatorname{argmin}} \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 \tag{95}$$

The optimal solution is $\mathbf{w}^\star = (\mathbf{X}^\top \mathbf{X})^{-1}\mathbf{X}^\top \mathbf{y}$. Looking closely at this optimal solution, it is easy to recognize that $\mathbf{w}^\star = \hat{\mathbf{R}}^{-1}\hat{\mathbf{r}}$, where $\hat{\mathbf{R}} = \frac{1}{m}(\mathbf{X}^\top \mathbf{X})$ is the empirical covariance matrix of the data and $\hat{\mathbf{r}} = \frac{1}{m}\mathbf{X}^\top \mathbf{y}$ the sample

cross-correlation vector. That is, the LS estimate is the solution of a data-fitting problem involving the empirical moments of data. In addition, it is important to recognize how problem (95) requires all the data $\mathbf{X}$ and $\mathbf{y}$ in advance.

Let us switch gears for a moment to a statistical setting by considering $y \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^N$ to be random variables with a certain probability distribution. The statistical counterpart of (95) is then to find the best parameter $\mathbf{w}$ minimizing the MSE:

$$\underset{\mathbf{w}}{\operatorname{argmin}}\, \mathbb{E}\{(y - \mathbf{w}^\top \mathbf{x})^2\} \tag{96}$$

where the expectation is over $\mathbf{x}$ and y. The optimal solution is $\mathbf{w}^\star = \mathbf{R}^{-1}\mathbf{r}$, where $\mathbf{R} = \mathbb{E}\{\mathbf{x}\mathbf{x}^\top\}$ is the covariance matrix of the data and $\mathbf{r} = \mathbb{E}\{y\mathbf{x}\}$ the sample cross-correlation vector. We can conclude that the solution of (95) is an approximation of the solution of the statistical problem (96).

If we have a good estimate $\hat{\mathbf{R}}$ of $\mathbf{R}$, and $\hat{\mathbf{r}}$ of $\mathbf{r}$, we can approach the optimal solution of (96) via iterative algorithms. For instance, by employing a gradient based method:

$$\begin{aligned} \mathbf{w}^{k+1} &= \mathbf{w}^k - \alpha \nabla f(\mathbf{w}^k) \\ &= \mathbf{w}^k - 2\alpha \mathbf{X}^\top (\mathbf{y} - \mathbf{X}\mathbf{w}^k) \\ &= \mathbf{w}^k + 2\alpha (\hat{\mathbf{R}}\mathbf{w}^k - \hat{\mathbf{r}}). \end{aligned} \tag{97}$$

For $m \to \infty$, i.e., for an increasing number of observation, the empirical covariance matrix $\hat{\mathbf{R}} \to \mathbf{R}$, as well as the sample cross-correlation vector $\hat{\mathbf{r}} \to \mathbf{r}$. In other words, the empirical moments converge to their statistical counterpart, with the consequence that $\lim_{k\to\infty} \mathbf{w}^k$ in (97) converge to the optimal solution of (96).

This is good news, however we made the strong assumption to have all the data $\mathbf{X}$ and $\mathbf{y}$ prior of the computation, not true in many applications. Envisioning an online setting, i.e. the case in which for each time instant k a new data $(\mathbf{x}_k, y_k)$ arrives on the fly, an alternative to solve (96) with a recursion of the form (97) isis by considering $\hat{\mathbf{R}} = \mathbf{x}_k \mathbf{x}_k^\top$ and $\hat{\mathbf{r}} = y_k \mathbf{x}_k$. That is, instead of waiting all the data to arrive and form the empirical covariance $\hat{\mathbf{R}}$ and correlation vector $\hat{\mathbf{r}}$, we approximate them with the current new data. The (estimate) of the gradient vector is then:

$$\hat{\nabla} f(\mathbf{w}^k) = -2y_k \mathbf{x}_k + 2\mathbf{x}_k \mathbf{x}_k^\top \mathbf{w}^k = -2\mathbf{x}_k (y_k - \mathbf{x}_k^\top \mathbf{w}^k). \tag{98}$$

The resulting gradient-based algorithm is known as Least Mean Square (LMS) algorithm:

$$\mathbf{w}^{k+1} = \mathbf{w}^k + 2\alpha \mathbf{x}_k (y_k - \mathbf{x}_k^\top \mathbf{w}^k) \tag{99}$$

On average, the LMS gradient direction approaches the optimal gradient direction, since for a fixed coefficient vector $\mathbf{w}^k$ we have: $\mathbb{E}[\hat{\nabla} f(\mathbf{w}^k)] = 2\mathbb{E}[\mathbf{x}_k \mathbf{x}_k^\top \mathbf{w}^k] - 2\mathbb{E}[y_k \mathbf{x}_k] = 2\mathbf{R}\mathbf{w}^k - 2\mathbf{r} = \nabla f(\mathbf{w}^k)$. That is, the expected value of the gradient $\hat{\nabla} f(\cdot)$ has the tendency to approach the true gradient $\nabla f(\cdot)$, and the solution of (99) approaches the one in (97).

.10.2. Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a stochastic approximation of gradient descent algorithm, where the actual gradient is replaced by an instantaneous estimate. Recall the general update rule for gradient descent is:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha^k \nabla f(\mathbf{w}^k) \tag{100}$$

where $f(\cdot)$ is the cost function we aim to minimize. Notice that our cost function is *data-dependent*, i.e. it depends on the parameter $\mathbf{w}$ as well as on the data $\mathscr{T}$, although we do not explicitly indicate it as $f(\mathbf{w}; \mathscr{T})$ to simplify the notation. This is typically the case when we solve an empirical risk minimization (ERM) problem (see Chapter **??**).

Stochastic gradient descent replaces the true gradient of $f(\cdot)$, evaluated taking into account all the data $\mathscr{T}$, with the approximate gradient obtained by using only one random sample $(\mathbf{x}_k, y_k)$ of the dataset. That is:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha^k \nabla f_k(\mathbf{w}^k), \quad k = 1, 2, \ldots \tag{101}$$

where $f_k(\mathbf{w}) := f(\mathbf{w}; (\mathbf{x}_k, y_k))$ is the iteration-dependent function using only the data $(\mathbf{x}_k, y_k)$. This is easy to implement because the gradient is now just the evaluation of the gradient at a single point. If samples are

chosen independently and with equal probability, then $\mathbb{E}\{\nabla f_k(\cdot)\} = \nabla f_k(\mathbf{w}^k)$, i.e., the stochastic gradient points in the right direction on average.

Convergence. Gradient descent converges because, for every iterate $\mathbf{w}^k$, the negative gradient $-\nabla f(\mathbf{w}^k)$ points towards the optimum $\mathbf{w}^\star$. Similarly, SGD converges because the negative stochastic gradients $-\nabla f_k(\mathbf{w}^k)$ do so on expectation; however it does not converge exactly to the optimum $\mathbf{w}^\star$ but hover around it, in a proportional manner to the step size and the variance of the stochastic gradients. In particular, for a fixed step size α:

$$\lim_{k\to\infty} \|\mathbf{w}^k - \mathbf{w}^\star\| \leq \mathcal{O}(\alpha) \tag{102}$$

i.e., the solution will hover in a radius of α.

Mini-batch gradient descent. Gradient descent considers all the samples $\{\mathbf{x}_n, y_n\} \in \mathcal{T}$ for the computation of the gradient, while SGD considers, at each iteration k, only one sample $(\mathbf{x}_k, y_k)$ randomly chosen from $\mathcal{T}$. An intermediate alternative is offered by selecting a batch of $Q < |\mathcal{T}|$ samples randomly chosen from $\mathcal{T}$, and compute the gradient by considering only those Q samples. I.e.,:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha^k \nabla f(\mathbf{w}; \mathcal{T}_Q), \quad k = 1, 2, \ldots \tag{103}$$

where $\mathcal{T}_Q$ contains the Q samples drawn from $\mathcal{T}$ at random. In this case,:

$$\lim_{k\to\infty} \|\mathbf{w}^k - \mathbf{w}^\star\| \leq \mathcal{O}(\alpha/\sqrt{Q}). \tag{104}$$

All in all, SGD is appealing because the iteration cost is independent of the number of samples m and can have big savings in terms of memory usage. However, several extensions of this algorithm have been proposed, trying to overcome its drawbacks.

.10.3. Extensions of Stochastic Gradient Descent

SGD is a simple algorithm for data driven optimization problems but it has a long learning process. The necessity to set a learning rate is one of the major drawbacks in developing an iterative descent algorithm, causing the algorithm to a possible divergence or to converge too slow. A simple yet useful extension is to let the learning step decrease when the number of iteration increases. To improve SGD, much work was proposed and we briefly discuss here a few popular alternatives.

Momentum. SGD with momentum employs an exponentially weighted moving average of the past gradients to update the current weights. The intuition is to prevents a possible zig-zagging behavior of the descent method, especially when the contour lines of the cost function are elliptical.

$$\mathbf{w}^{k+1} := \mathbf{w}^k - \alpha \nabla f_k(\mathbf{w}^k) + \gamma \Delta \mathbf{w}^k, \tag{105}$$

where α is the step-size and γ is the *momentum factor* (friction), which has been empirically demonstrated to be optimal for 0.9. Notice in (105) that the second term $-\alpha \nabla f_k(\mathbf{w}^k)$ captures the gradient at iteration k, while the third term $\gamma \Delta \mathbf{w}^k$ is the previous search direction that is considered also for the update at iteration $k+1$. As mentioned above, to avoid large oscillations when $\mathbf{w}^{k+1}$ gets close to the minima, an adjusted step-size α is considered in each iteration of the form $\alpha^k = \alpha^0/(1 + dk)$, where d is a decay factor and α^0 is the starting step-size.

So far, the descent methods used the same learning rate α for all the parameters w_j in the vector parameter $\mathbf{w}$.

AdaGrad. The Adaptive Gradient algorithm (AdaGrad) uses a different learning rate for each entry w_j of vector $\mathbf{w}$ at every iteration k. Given the gradient at iteration k, $\nabla f(\mathbf{w}^k)$, with partial derivative of f w.r.t. the parameter w_j in its j-th entry, i.e., $[\nabla f(\mathbf{w}^k)]_j = \partial f(\mathbf{w}^k)/\partial w_j$, the per-parameter update is:

$$w_j^{k+1} = w_j^k - \frac{\alpha}{\sqrt{\sum_{t=1}^k \frac{\partial f(\mathbf{w}^t)}{\partial w_j}^2 + \epsilon}} \cdot \frac{\partial f(\mathbf{w}^k)}{\partial w_j}. \tag{106}$$

That is, Adagrad is a standard GD update, where the learning rate for each parameter w_j depends from the history of the gradients for the parameter up to the current iteration. In vector form, we have:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha \operatorname{diag}(\mathbf{G}_k)^{-\frac{1}{2}} \odot \nabla f(\mathbf{w}^k), \tag{107}$$

where $\odot$ is the Hadamard (entry-wise) product and $\mathbf{G}_k = \sum_{t=1}^{k} \nabla f(\mathbf{w}^k) \nabla f(\mathbf{w}^k)^\top$ is the matrix containing on the main diagonal the sum of squares of all the partial derivative up to iteration k.

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate; indeed, many implementations use a default value of $\alpha = 0.01$. Adagrad's main weakness is its accumulation of the squared gradients in the denominator, making the learning rate tend to zero, resulting in ineffective parameter update.

RMSProp. The Root Mean Square Propagation (RMSProp) is also a method in which the learning rate is adapted for each of the parameters. In particular, it tries to overcome the diminishing learning rate of Adagrad.

$$w_j^{k+1} = w_j^k - \frac{\alpha}{\sqrt{\beta(\sum_{t=1}^{k-1} \frac{\partial f(\mathbf{w}^t)}{\partial w_j}^2) + (1-\beta)\frac{\partial f(\mathbf{w}^k)}{\partial w_j}^2}} \cdot \frac{\partial f(\mathbf{w}^k)}{\partial w_j} \tag{108}$$

where β is a forgetting factor.

Adam. The adaptive moment estimation (Adam) [3] combines momentum and RMSProp. In addition to storing an exponentially decaying average of past squared gradients like RMSProp, Adam also keeps an exponentially decaying average of past gradients, similar to the momentum method. That is:

$$\mathbf{m}_k = \beta_1 \mathbf{m}_{k-1} + (1-\beta_1)\nabla f(\mathbf{w}^k) \tag{109}$$

is the exponentially moving average of the past gradients at iteration k, where $\beta_1 \in (0,1)$ is the exponential decay rate. Likewise:

$$\mathbf{v}_k = \beta_2 \mathbf{v}_{k-1} + (1-\beta_2)\nabla f(\mathbf{w}^k)^{\circ 2} \tag{110}$$

is the decaying average of past squared gradients at iteration k, where $^\circ$ indicate the element-wise power and $\beta_2 \in (0,1)$ is the exponential decay rate. Because (109) and (110) are biased towards zero, the following corrections are often performed:

$$\hat{\mathbf{m}}_k = \frac{\mathbf{m}_k}{1-\beta_1^k} \qquad \hat{\mathbf{v}}_k = \frac{\mathbf{v}_k}{1-\beta_2^k} \tag{111}$$

Finally the update of the parameter is computed as:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \alpha \frac{\hat{\mathbf{m}}_k}{\sqrt{\hat{\mathbf{v}}_k} + \epsilon}. \tag{112}$$

Typical choice of β_1, β_2 are 0.9 and 0.999, respectively, while the step-size α can also be iteration-dependent. The advantage of Adam is its ease of implementation, the computational efficiency and little memory requirements. All this, together with the ability to cope with problems that are large in terms of data and/or parameters, makes Adam a robust and well-suited algorithm to a wide range of (non-convex) optimization problems in the field machine learning.

Non-convex functions: local minima. In non-convex functions, we (may) end up in local optima. This is especially true when then parameter dimension is very high. It turns out that SGD and its variants help in preventing the algorithm to being stacked in local optima, or better, in saddle points, or plateaus (i.e., surfaces of the cost function where the gradient is almost zero). Because of the random gradient perturbation, SGD often escapes from the plateaus and reach a"satisfactory" local minima.

.11. Conclusions and Further Readings

In this chapter, we discussed how to solve optimization problems with iterative methods. In Section .9.1 we introduced gradient descent, the "vanilla" version of all the other algorithms covered all over the chapter, and in Section .9.2 we showed, together with the Jacobi method and conjugate gradient method, how to use it to solve linear systems of equations. In Section .10 we described how iterative algorithms can be used in data-driven settings to solve optimization problems. Finally, we introduced the stochastic gradient descent as a way to cope with the high problem dimensionality, and illustrated some of its variants, such as AdaGrad and Adam, which mitigate its drawbacks. For further study on the subject, we recommend the following references.

Bibliography

[1] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[2] Marc Peter Deisenroth, A Aldo Faisal, and Cheng Soon Ong. *Mathematics for machine learning*. Cambridge University Press, 2020.

[3] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[4] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[5] Jonathan Richard Shewchuk et al. *An introduction to the conjugate gradient method without the agonizing pain*. 1994.

[6] Shiliang Sun et al. "A survey of optimization methods from a machine learning perspective". In: *IEEE transactions on cybernetics* 50.8 (2019), pp. 3668–3681.