Build version as of 2018-10-08 11:01:45
Revision number 332

On

Convolution of Graph Signals

And

Deep Learning on Graph Domains

Abstract

This manuscript is a thesis submitted to apply for a doctorate. It is devoted to two subjects. The first one is about extending the discrete convolution to graph signals. The second one is about extending neural networks to graph domains. Both subjects are related since neural networks can make use of convolutions to leverage the underlying structure of their input domain.

Résumé

Ce manuscrit est une thèse soumise pour candidater au grade de docteur. Il est dévolu à deux sujets. Le premier traite d'extensions de la convolution discrète aux signaux sur graphe. Le second traite d'extensions de l'ensemble de définition des réseaux de neurones à des graphes. Les deux sujets sont reliés car les réseaux de neurones peuvent tirer profit de la structure sous-jacente de leur ensemble de définition à l'aide de convolutions.

Contents

Introduction

One of the first appearances of the *convolution* operation was in the eighteenth century (D'Alembert, 1754, according to Dominguez-Torres, 2010). Since then it has been used in a wide range of domains, including applied mathematics, physics, engineering, informatics and real world problems. In today's era of computerized industry and big data, the convolution has never been more useful. A famous example is its usage in deep learning algorithms for image processing (LeCun et al., 2015). But this is just the tip of the iceberg. Years after years, IT companies acquire more and more data. With hashing algorithms, accessing single points or moderately sized batches of data is relatively easy. However, processing the entire database's knowledge is another story. Algorithms that do not scale well are too time consuming, so only those that traverse the entire database only a few times remain feasible. And that is the point: traversing the database (done in a parallelized manner) to process it can be modelized with a convolution. Therefore this little mathematical tool is bound to play a great role! An example can be seen in the company that funded this Ph.D.: one of its technologies for processing large quantities of data is a programming language centered around a few frameworks. One of them is nothing more than a reimplemention of the convolution by various complicated functions, which helps tremendously to produce simplified scripts.

The subject of this thesis is a timely one, since deep learning have never received as much spotlight as in the last decade. However, deep learning

models are often very specialized to their use cases. Standard Convolutional Neural Networks (CNNs) can only be applied to datasets for which each object can be modelized by a signal defined on an Euclidean domain, like images or sounds. This is because the discrete convolution takes into account the structure of the domain of its inputs, in addition to raw data. Our goal is to study and find ways to extend deep learning models to signals defined on a broader range of domains. To this end we build an algebraic theory of convolutions of graph signals, with the hope to characterize what a more generic definition of convolution should be and what properties it should preserve to keep its usefulness in deep learning models. We choose to modelize the domain of signals under study with a graph, since this structure can represent a wide range of domains. Our subject fits the idea that efforts in the Artificial Intelligence (AI) field should tend toward a general-purpose AI, and in a lesser extent, that AI-based algorithm, like deep learning, should seek genericity.

> *The ultimate aim is to use these general-purpose technologies and apply them to all sorts of important real world problems.*
>
> — *Demis Hassabis*

This manuscript is broken down into three chapters. Each chapter is preceded by a short overview so that the reader can grasp its essential contents at a glance. In Chapter 1, we present our domains of interest with a selected literature review. Then, in Chapter 2, we theorize an algebraic understanding of convolutions of graph signals. Finally, in Chapter 3, we study neural networks intended for graph domains.

Chapter 1
Presentation of the field

Contents

Chapter overview

In this chapter, we present notions related to our domains of interest. One of them, *deep learning*, is the field of research that focuses on a particular class of functions: *neural networks*. Since we try to employ a rigorous approach, we first define properly their input domain and their codomain, which can be modelized as *tensor spaces*. In particular, we give original definitions of tensors in Section 1.1 that are appropriate for the study of neural networks. We also explain how data is handled and manipulated. We give definitions of some binary operations that are important for our study: *tensor contraction*, and *convolution*. In Section 1.2, we define neural networks, discuss their biological interpretation, present how they learn, and relate some historical advances. Then we introduce common layers, especially *convolutional* ones for which we demonstrate a little result that helps toward our study. In the last section, Section 1.3, we present the field of *deep learning on graphs*. We start with definitions about graphs and signals, and then we describe use cases. Finally, we give a review of state-of-the-art models in two separate subsections, one on spectral methods, and the other one on vertex-domain methods.

1.1 Tensors

Intuitively, tensors in the field of deep learning are defined as a generalization of vectors and matrices, as if vectors were tensors of rank 1 and matrices were tensors of rank 2. That is, they are objects in a vector space and their dimensions are indexed using as many indices as their rank, so that they can be represented by multidimensional arrays. In mathematics, a tensor can be defined as a special type of multilinear function (Bass, 1968; Marcus, 1975; Williamson, 2015) which can be represented by a multidimensional array. Alternatively, Hackbush proposes a mathematical construction of a tensor space as a quotient set of the span of an appropriately defined tensor product (Hackbusch, 2012), which coordinates in a basis can also be represented by a multidimensional array. In particular in the field of mathematics, tensors enjoy an intrinsic definition that neither depends on a representation nor would change the underlying object after a change of basis, whereas in our domain, tensors are confounded with their representation.

1.1.1 Definition

Our definition of tensors is such that they are a bit more than multidimensional arrays but not as much as mathematical tensors. They are embedded in a vector space, called tensor space, so that deep learning objects can be later defined rigorously.
Given canonical bases, we first define a tensor space, then we relate it to the definition of the tensor product of vector spaces.

Definition 1. Tensor space
We define a *tensor space* $\mathbb{T}$ of rank r as a vector space such that its canonical basis is a Cartesian product of the canonical bases of r finite-dimensional vector spaces.

Its shape is denoted $n_1 \times n_2 \times \cdots \times n_r$, where the $\{n_k\}$ are the dimensions of the vector spaces.

Remark. Unless stated otherwise, vector spaces are assumed to be over the field of real numbers $\mathbb{R}$.

Definition 2. Tensor product of vector spaces
Given r vector spaces $\mathbb{V}_1, \mathbb{V}_2, \ldots, \mathbb{V}_r$, their *tensor product* is the tensor space $\mathbb{T}$ spanned by the Cartesian product of their canonical bases under coordinate-wise sum and outer product.
We use the notation $\mathbb{T} = \bigotimes_{k=1}^{r} \mathbb{V}_k$.

Remark. This simpler definition is indeed equivalent with the definition of the tensor product given in (Hackbusch, 2012, p. 51). The drawback of our definition is that it depends on the canonical bases, which at first can seem limiting as being canonical implies that they are bounded to a certain system of coordinates. However this is not a concern in our domain as we need not distinguish tensors from their representation.

Naming convention
We will also call *vector space* a tensor space of rank 1. In case there is a vector space that we do not need to see as a tensor space of rank 1, we may use the term *linear space* instead. We also make a clear distinction between the terms *dimension* (that is, for a tensor space it is equal to $\prod_{k=1}^{r} n_k$) and the term *rank* (equal to r). Note that some authors use the term *order* or *mode* instead of *rank* as the latter is also affected to another notion.

Definition 3. Tensor
A *tensor* t is an object of a tensor space. The *shape* of t, which is the same as the shape of the tensor space it belongs to, is denoted $n_1^{(t)} \times n_2^{(t)} \times \cdots \times n_r^{(t)}$.

1.1.2 Manipulation

In this subsection, we describe notations and operators used to manipulate data stored in tensors. The formalism we present here should be familiar to the researcher in the domain, since it is similar to the notations used by NumPy (Oliphant, 2006) and most deep learning libraries *e.g.* TensorFlow (Abadi et al., 2015), PyTorch (Paszke et al., 2017), Mxnet (Chen et al., 2015), Keras (Chollet et al., 2015).

Definition 4. Indexing
An *entry* of a tensor $t \in \mathbb{T}$ is one of its scalar coordinates in the canonical basis, denoted $t[i_1, i_2, \ldots, i_r]$.
More precisely, if $\mathbb{T} = \bigotimes_{k=1}^{r} \mathbb{V}_k$, with bases $((e_k^i)_{i=1,\ldots,n_k})_{k=1,\ldots,r}$, then we have

$$t = \sum_{i_1=1}^{n_1} \cdots \sum_{i_r=1}^{n_r} t[i_1, i_2, \ldots, i_r](e_1^{i_1}, \ldots, e_r^{i_r})$$

We call the *index space* of $\mathbb{T}$ the Cartesian product of integer intervals $\mathcal{I} = \prod_{k=1}^{r} [\![1, n_k]\!]$.

Remark. When using an index i_k for an entry of a tensor t, we implicitly assume that $i_k \in [\![1, n_k^{(t)}]\!]$ unless otherwise specified.

Definition 5. Subtensor
A *subtensor* t' is a tensor of same rank composed of entries of t. We denote

$$\begin{cases} t' = t[[i^1_1, \dots, i^1_{n_1(t')}], \dots, [i^r_1, \dots, i^r_{n_r(t')}]] \\ \text{where } t'[j_1, \dots, j_r] = t[i^1_{j_1}, \dots, i^r_{j_r}] \end{cases}$$

Given p, if $[i^p_1, \dots, i^p_{n_p(t')}]$ is a singleton, we drop the brackets. If it is a strictly increasing contiguous sequence, we write instead $i^p_1 : i^p_{n_p(t')}$. We drop the left (or right) of : if it is the possible lower (or upper) bound.

Remark. This notation for indexing subtensors is not the same as the one used by NumPy. However the use of : is similar (except that we also include the right of :).

Definition 6. Slicing
A *slice* operation, along the last ranks $\{r_1, r_2, \dots, r_s\}$, and indexed by $(i_{r_1}, i_{r_2}, \dots, i_{r_s})$, is a morphism $s : \mathbb{T} = \bigotimes_{k=1}^{r} \mathbb{V}_k \to \bigotimes_{k=1}^{r-s} \mathbb{V}_k$, such that:

$$s(t)[i'_1, i'_2, \dots, i'_{r-s}] = t[i'_1, i'_2, \dots, i'_{r-s}, i_{r_1}, i_{r_2}, \dots, i_{r_s}]$$
$$\textit{i.e.} \quad s(t) := t[:, :, \dots, :, i_{r_1}, i_{r_2}, \dots, i_{r_s}]$$

where $:=$ means that entries of the right operand are assigned to the left operand. We denote $t_{i_{r_1}, i_{r_2}, \dots i_{r_s}}$ and call it the *slice* of t. Slicing along a subset of ranks that are not the lasts is defined similarly. $s(\mathbb{T})$ is called a *slice subspace*.

Definition 7. Flattening
A *flatten* operation is an isomorphism $f : \mathbb{T} \to \mathbb{V}$, between a tensor space $\mathbb{T}$ of rank r and an n-dimensional vector space $\mathbb{V}$, where $n = \prod_{k=1}^{r} n_k$. It is

characterized by a bijection in the index spaces $g : \prod_{k=1}^{r} [\![1, n_k]\!] \to [\![1, n]\!]$ such that

$$\forall t \in \mathbb{T}, f(t)[g(i_1, i_2, \ldots, i_r)] = f(t[i_1, i_2, \ldots, i_r])$$

We call an inverse operation a *de-flatten* operation.

Row major ordering

The choice of g determines in which order the indexing is made. g is reminiscent of how data of multidimensional arrays or tensors are stored internally by programming languages. In most tensor manipulation languages, incrementing the memory address (*i.e.* the output of g) will first increment the last index i_r if $i_r < n_r$ (and if else $i_r = n_r$, then $i_r := 1$ and ranks are ordered in reverse lexicographic order to decide what indices are incremented). This is called *row major ordering*, as opposed to *column major ordering*. That is, in row major, g is defined as

$$g(i_1, i_2, \ldots, i_r) = \sum_{p=1}^{r} \left(\prod_{k=p+1}^{r} n_k \right) i_p \tag{1}$$

Definition 8. Reshaping

A *reshape* operation is an isomorphism defined on a tensor space $\mathbb{T} = \bigotimes_{k=1}^{r} \mathbb{V}_k$ such that some of its basis vector spaces $\{\mathbb{V}_k\}$ are de-flattened and some of its slice subspaces are flattened.

1.1.3 Binary operations

We define binary operations on tensors that we'll later have use for. In particular, we define *tensor contraction* which is sometimes called *tensor multiplication*, *tensor product* or *tensor dotproduct* by other sources. We also define *convolution* and *pooling* which serve as the common building blocks of convolution neural network architectures.

Definition 9. Contraction
A *tensor contraction* between two tensors, along ranks of same dimensions, is defined by natural extension of the dot product operation to tensors. More precisely, let $\mathbb{T}_1$ a tensor space of shape $n_1^{(1)} \times n_2^{(1)} \times \cdots \times n_{r_1}^{(1)}$, and $\mathbb{T}_2$ a tensor space of shape $n_1^{(2)} \times n_2^{(2)} \times \cdots \times n_{r_2}^{(2)}$, such that $\forall k \in [\![1, s]\!], n_{r_1-(s-k)}^{(1)} = n_k^{(2)}$, then the tensor contraction between $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$ is defined as:

$$\begin{cases} t_1 \otimes t_2 = t_3 \in \mathbb{T}_3 \text{ of shape } n_1^{(1)} \times \cdots \times n_{r_1-s}^{(1)} \times n_{s+1}^{(2)} \times \cdots \times n_{r_2}^{(2)} \text{ where} \\ t_3[i_1^{(1)}, \ldots, i_{r_1-s}^{(1)}, i_{s+1}^{(2)}, \ldots, i_{r_2}^{(2)}] = \\ \sum_{k_1=1}^{n_1^{(2)}} \cdots \sum_{k_s=1}^{n_s^{(2)}} t_1[i_1^{(1)}, \ldots, i_{r_1-s}^{(1)}, k_1, \ldots, k_s]\, t_2[k_1, \ldots, k_s, i_{s+1}^{(2)}, \ldots, i_{r_2}^{(2)}] \end{cases}$$

For the sake of simplicity, we omit the case where the contracted ranks are not the last ones for t_1 and the first ones for t_2. But this definition still holds in the general case subject to a permutation of the indices.

Definition 10. Covariant and contravariant indices
Given a tensor contraction $t_1 \otimes t_2$, indices of the left hand operand t_1 that are not contracted are called *covariant* indices. Those that are contracted are called *contravariant* indices. For the right operand t_2, the naming convention is the opposite. The set of covariant and contravariant indices of both operands are called the *transformation laws* of the tensor contraction.

Remark. Contrary to most mathematical definitions, tensors in deep learning are independent of any transformation law, so that they must be specified for tensor contractions.

Einstein summation convention

The Einstein summation convention is a notational convention to write a sum-product expression as a product expression. The summation indices are those that appear simultaneously in the superscript of the left operand and in the subscript of the right one, if subscripts precede superscripts in the notation, or else vice-versa. For example, a dot product is written $u_k v^k = \lambda$ and a matrix product is written ${A_i}^k {B_k}^j = {C_i}^j$.

The tensor contraction of Definition 9 can be rewritten using this convention:

$$ {t_1}_{i_1^{(1)} \cdots i_{r_1-s}^{(1)}}{}^{k_1 \cdots k_s} {t_2}_{k_1 \cdots k_s}{}^{i_{s+1}^{(2)} \cdots i_{r_2}^{(2)}} = {t_3}_{i_1^{(1)} \cdots i_{r_1-s}^{(1)}}{}^{i_{s+1}^{(2)} \cdots i_{r_2}^{(2)}} \tag{2} $$

Proposition 11. A contraction can be rewritten as a matrix product.

Proof. Using notation of (2), with the reshapings $t_1 \mapsto T_1$, $t_2 \mapsto T_2$ and $t_3 \mapsto T_3$ defined by grouping all covariant indices into a single index and all contravariant indices into another single index, we can rewrite

$$ {T_1}_{g_i(i_1^{(1)}, \ldots, i_{r_1-s}^{(1)})}{}^{g_k(k_1, \ldots, k_s)} {T_2}_{g_k(k_1, \ldots, k_s)}{}^{g_j(i_{s+1}^{(2)}, \ldots, i_{r_2}^{(2)})} = {T_3}_{g_i(i_1^{(1)}, \ldots, i_{r_1-s}^{(1)})}{}^{g_j(i_{s+1}^{(2)}, \ldots, i_{r_2}^{(2)})} $$

where g_i, g_k and g_j are bijections defined similarly as in (1). □

Definition 12. Convolution

The *n-dimensional convolution*, denoted $*^n$, between $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$, where $\mathbb{T}_1$ and $\mathbb{T}_2$ are of the same rank n such that $\forall p \in [\![1, n]\!], n_p^{(1)} \geq n_p^{(2)}$,

is defined as:

$$\begin{cases} t_1 *^n t_2 = t_3 \in \mathbb{T}_3 \text{ of shape } n_1^{(3)} \times \cdots \times n_n^{(3)} \text{ where} \\ \forall p \in [\![1, n]\!], n_p^{(3)} = n_p^{(1)} - n_p^{(2)} + 1 \\ t_3[i_1, \ldots, i_n] = \sum_{k_1=1}^{n_1^{(2)}} \cdots \sum_{k_n=1}^{n_n^{(2)}} t_1[i_1 + n_1^{(2)} - k_1, \ldots, i_n + n_n^{(2)} - k_n]\, t_2[k_1, \ldots, k_n] \end{cases}$$

Proposition 13. A convolution can be rewritten as a matrix product.

Proof. Let $t_1 *^n t_2 = t_3$ defined as previously with $\mathbb{T}_1 = \bigotimes_{k=1}^{r} \mathbb{V}_k^{(1)}$, $\mathbb{T}_2 = \bigotimes_{k=1}^{r} \mathbb{V}_k^{(2)}$. Let $t_1' \in \bigotimes_{k=1}^{r} \mathbb{V}_k^{(1)} \otimes \bigotimes_{k=1}^{r} \mathbb{V}_k^{(2)}$ such that $t_1'[i_1, \ldots, i_n, k_1, \ldots, k_n] = t_1[i_1 + n_1^{(2)} - k_1, \ldots, i_n + n_n^{(2)} - k_n]$, then

$$t_3[i_1, \ldots, i_n] = \sum_{k_1=1}^{n_1^{(2)}} \cdots \sum_{k_n=1}^{n_n^{(2)}} t_1'[i_1, \ldots, i_n, k_1, \ldots, k_n]\, t_2[k_1, \ldots, k_n]$$

where we recognize a tensor contraction. Proposition 11 concludes. □

The two following operations are meant to further decrease the shape of the resulting output.

Definition 14. Strided convolution
The n-dimensional *strided* convolution, with strides $s = (s_1, s_2, \ldots, s_n)$, denoted $*_s^n$, between $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$, where $\mathbb{T}_1$ and $\mathbb{T}_2$ are of the same rank n such that $\forall p \in [\![1, n]\!], n_p^{(1)} \geq n_p^{(2)}$, is defined as:

$$\begin{cases} t_1 *_s^n t_2 = t_4 \in \mathbb{T}_4 \text{ of shape } n_1^{(4)} \times \cdots \times n_n^{(4)} \text{ where} \\ \forall p \in [\![1, n]\!], n_p^{(4)} = \lfloor \frac{n_p^{(1)} - n_p^{(2)} + 1}{s_p} \rfloor \\ t_4[i_1, \ldots, i_n] = (t_1 *^n t_2)[(i_1 - 1)s_n + 1, \ldots, (i_n - 1)s_n + 1] \end{cases}$$

Remark. Informally, a strided convolution is defined as if it were a regular subsampling of a convolution. They match if $s = (1, 1, \ldots, 1)$.

Definition 15. Pooling
Let a real-valued function f defined on all tensor spaces of any shape, *e.g.* the *max* or *average* function. An f-pooling operation is a mapping $t \mapsto t'$ such that each entry of t' is an image by f of a subtensor of t.

Remark. Usually, the set of subtensors that are reduced by f into entries of t' are defined by a regular partition of the entries of t.

1.2 Deep learning

In this manuscript, we adopt the point of view that a neural network is first a mathematical function, even though it derives its name from biological inspiration. That is, we won't discuss whether any of our works are biologically plausible or not, but we may provide biological interpretation when it happens.
In this section, we present a mathematical formalization and its biological interpretation. Then, we review a few important advances in the field before we finally present the most commonly used layers.

1.2.1 Neural networks

A feed-forward neural network could originally be formalized as a composite function chaining linear and non-linear functions (Rumelhart et al., 1985; LeCun et al., 1989; LeCun, Bengio, et al., 1995). That was still the case in 2012 when important breakthroughs regenerated a surge of interest in the field (Hinton et al., 2012; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). However, in more recent years, more complex architectures have emerged (Szegedy et al., 2015; He et al., 2016a; Zoph and Le, 2016; Huang et al., 2017), such that the former formalization does not suffice. We provide a definition for the first kind of neural networks (Definition 16) and use it to present its related concepts. Then we give a more generic definition (Definition 20).
Note that in this manuscript, we only consider neural networks that are *feed-forward* (Zell, 1994; Wikipedia, 2018a), as opposed to *recurrent*.
We denote by I_f the *domain of definition* of a function f ("I" stands for "input") and by $O_f = f(I_f)$ its *image* ("O" stands for "output"), and we represent it as $I_f \xrightarrow{f} O_f$ or $f : I_f \to O_f$.

Definition 16. Neural network (simply connected)
Let f be a function such that I_f and O_f are vector or tensor spaces.
f is a *(simply connected) neural network function* if there are a series of affine functions $(g_k)_{k=1,2,..,L}$ and a series of non-linear derivable univariate functions $(h_k)_{k=1,2,..,L}$ such that:

$$\begin{cases} \forall k \in [\![1, L]\!], f_k = h_k \circ g_k, \\ I_f = I_{f_1} \xrightarrow{f_1} O_{f_1} \cong I_{f_2} \xrightarrow{f_2} \dots \xrightarrow{f_L} O_{f_L} = O_f, \\ f = f_L \circ ... \circ f_2 \circ f_1 \end{cases}$$

The couple (g_k, h_k) is called the *k-th layer* of the neural network. L is its depth. For $x \in I_f$, we denote by $x_k = f_k \circ ... \circ f_2 \circ f_1(x)$ the *activations* of the k-th layer. We denote by $\mathcal{N}$ the set of neural network functions.

Definition 17. Activation function
An *activation function* h is a real-valued univariate function that is non-linear and derivable, that is also defined by extension with the functional notation $h(v)[i] = h(v[i])$.

Definition 18. Layer
A layer is a couple $\mathcal{L} = (g, h) : I \to O$, where $g : I \to O$ is a linear function, and $h : O \to O$ is an activation function. It computes the function

$$y = h(g(x) + b)$$

where b is a constant called *bias*.

That is, in the simple formalization, a neural network is just a sequence of layers.

Remark. The bias augments the expressivity of the layers. For notational convenience, we may sometimes omit to write it down.

The most common activation function is the *rectified linear unit* (ReLU) (Glorot et al., 2011), used for its better practical performances and faster computation times. It implements the *rectifier* function $h : x \mapsto max(0, x)$ (with convention $h'(0) = 0$), as depicted on Figure 1.

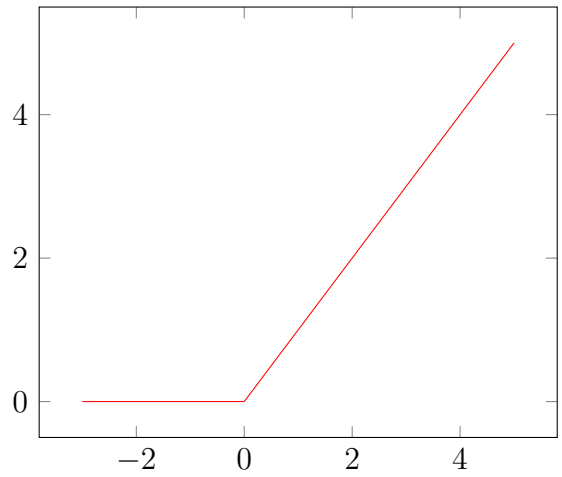


Figure 1: ReLU activation function

Examples

Let $f : x \to y$ be a neural network. For example, if f is used to classify its input x in one of c classes, then its output y would be a vector of dimension c, and each dimension corresponds to a class. The prediction of f for the class of x is the dimension of y where it has the bigger value. Typically, f is terminated by a softmax activation (Wikipedia, 2018b), so that values of the output y fall in the range $[0, 1]$, and so that y tends to have a dimension with a much bigger weight as to facilitates discrimination.
A neural network that comprises convolutional layers, *i.e.* layers *s.t.* g is expressed with a convolution, is called a Convolutional Neural Network (CNN). An old example is the LeNet-5 architecture (LeCun et al., 1989) as

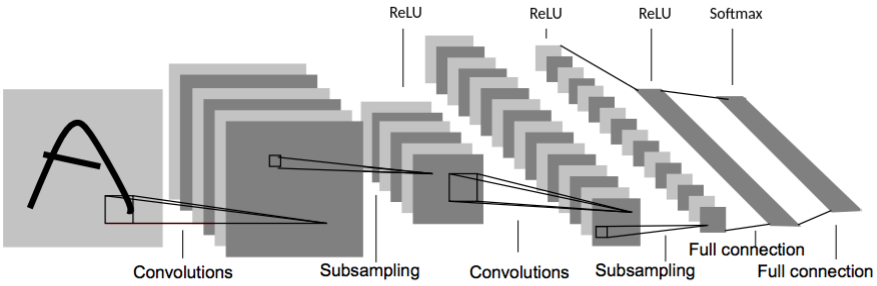


Figure 2: LeNet-5 (LeCun et al., 1989)

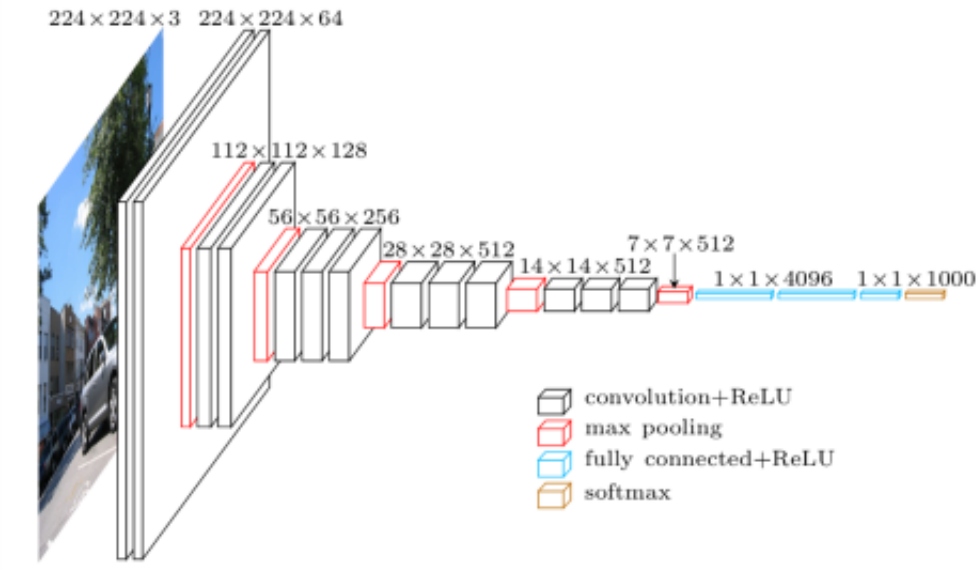


Figure 3: VGG-16 (Simonyan and Zisserman, 2014, figure from Cord, 2016)

depicted in Figure 2. It implements a function

$$f = h_4 \circ g_4 \circ \cdots \circ h_1 \circ g_1$$

where g_1 and g_2 are linear functions that applies 5x5 convolutions followed by subsampling, h_1, h_2 and h_3 are ReLU activations, and h_4 is a softmax activation. It was originally applied to the task of handwritten digit classifications (for example for automatically reading postal ZIP codes).
Another example is the VGG architecture, a very deep CNN, and was state-of-the-art in image classification in 2014 (Simonyan and Zisserman). It is depicted on Figure 3. In more recent years, state-of-the-art architectures can no longer be described with a simple formalization.
The former neural networks are said to be *simply connected* because each layer only takes as input the output of the previous one. We give a more general definition after first defining branching operations.

Definition 19. Branching
A *binary branching operation* between two tensors, $x_{k_1} \bowtie x_{k_2}$, outputs, subject to shape compatibility, either their addition, either their concatenation along a rank, or their concatenation as a list.

Definition 20. Neural network (generic definition)
The set of *neural network* functions $\mathcal{N}$ is defined inductively as follows

1. $Id \in \mathcal{N}$
2. $f \in \mathcal{N} \wedge (g, h)$ is a layer $\wedge\, O_f \subset I_g \Rightarrow h \circ g \circ f \in \mathcal{N}$
3. for all shape compatible branching operations:
   $f_1, f_2, \ldots, f_n \in \mathcal{N} \Rightarrow f_1 \bowtie f_2 \bowtie \cdots \bowtie f_n \in \mathcal{N}$

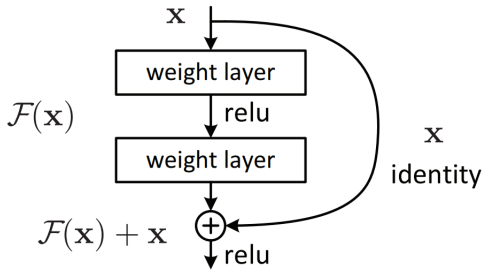


Figure 4: Module with a residual connection (He et al., 2016a)

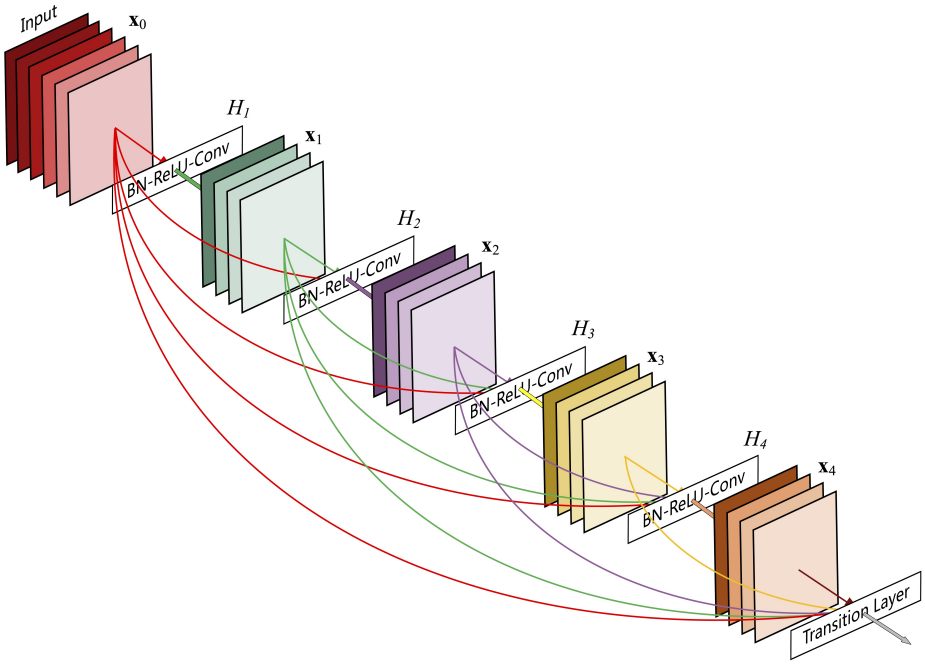


Figure 5: DenseNet (Huang et al., 2017)

Examples

The neural network proposed in (Szegedy et al., 2015), called *Inception*, use depth-wise concatenation of feature maps. Residual networks (ResNets, He et al., 2016a) make use of *residual connections*, also called *skip connections*, *i.e.* an activation that is used as input in a lower level is added to another activation at an upper level, as depicted on Figure 4. Densely connected networks (DenseNets, Huang et al., 2017) have their activations concatenated with all lower level activations. These neural networks had demonstrated state of the art performances on the imagenet classification challenge (Deng et al., 2009), outperforming simply connected neural networks. For example, DenseNet is depicted on Figure 5.

Remark. For layer indexing convenience, we still use the simple formalization in the subsequent subsections, even though the presentation would be similar with the generic formalization.

1.2.2 Interpretation

Until now, we have formally introduced a neural network as a mathematical function. As its name suggests, such function can be indeed interpreted from a connectivity perspective (LeCun, 1987).

Definition 21. Connectivity matrix
Let g a linear function. Without loss of generality subject to a flattening, let's suppose I_g and O_g are vector spaces. Then there exists a *connectivity matrix* W_g, such that:

$$\forall x \in I_g, g(x) = W_g x$$

We denote W_k the connectivity matrix of the k-th layer.

Biological inspiration

A *neuron* is defined as a computational unit that is biologically inspired (McCulloch and Pitts, 1943). Each neuron is capable of:

1. receiving modulated signals from other neurons and aggregate them,
2. applying to the result an activation function,
3. passing the signal to other neurons.

That is to say, each domain $\{I_{f_k}\}$ and O_f can be interpreted as a layer of neurons, with one neuron for each dimension. The connectivity matrices $\{W_k\}$ describe the connections between each successive layers. A neuron is illustrated on Figure 6.

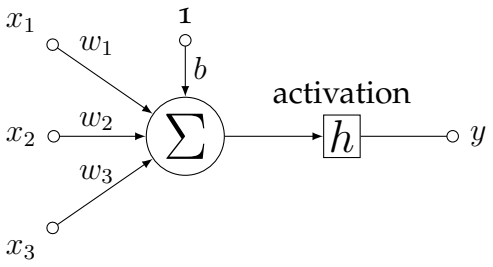


Figure 6: The McCulloch-Pitts model of a neuron

1.2.3 Training

Given an objective function F, training is the process of incrementally modifying a neural network f upon obtaining a better approximation of F. The most used training algorithms are based on gradient descent, as proposed in (Widrow and Hoff, 1960). These algorithms became popular since (Rumelhart et al., 1985). Informally, f is parameterized with initial weights that characterize its linear parts. These weights are modified step by step. At each step, a batch of samples are fed to the network, and their approximation errors sum to a loss. The weights of the network are updated in the opposite direction to their gradient with respect to that

loss. If the samples are shuffled and grouped in batches, this is called *Stochastic* gradient descent (SGD). Stochastic approximation (Robbins and Monro, 1985) tends to minimize effects of outliers on the training and is agnostic of the order in which the samples are fed.

Definition 22. Weights
Let consider the k-th layer of a neural network f. We define its weights as coordinates of a vector θ_k, called the *weight kernel*, such that:

$$\forall (i,j), \begin{cases} \exists p, W_k[i,j] := \theta_k[p] \\ \text{or } W_k[i,j] = 0 \end{cases}$$

A weight p that appears multiple times in W_k is said to be *shared*. Two parameters of W_k that share a same weight p are said to be *tied*. The number of weights of the k-th layer is $n_1^{(\theta_k)}$.

Learning
A *loss* function $\mathcal{L}$ penalizes the output $x_L = f(x)$ relatively to the approximation error $|f(x) - F(x)|$. Gradient w.r.t. θ_k, denoted $\vec{\nabla}_{\theta_k}$, is used to update the weights via an optimization algorithm based on gradient descent and a learning rate α, that is:

$$\theta_k^{(\text{new})} = \theta_k^{(\text{old})} - \alpha \cdot \vec{\nabla}_{\theta_k} \left(\mathcal{L}\left(x_L, \theta_k^{(\text{old})}\right) + \mathcal{R}\left(\theta_k^{(\text{old})}\right) \right) \tag{3}$$

where $\mathcal{R}$ is a regularizer, and where α can be a scalar or a vector and $\cdot$ can denote outer or coordinate-wise product, depending on the optimization algorithm that is used.

Linear complexity
Without loss of generality, we assume that the neural network is simply connected. Thanks to the chain rule, $\vec{\nabla}_{\theta_k}$ can be computed using gradients that are w.r.t. x_k, denoted $\vec{\nabla}_{x_k}$, which in turn can be computed using

gradients w.r.t. outputs of the next layer $k+1$, up to the gradients given on the output layer.
That is:

$$\begin{aligned} \vec{\nabla}_{\theta_k} &= J_{\theta_k}(x_k)\vec{\nabla}_{x_k} \\ \vec{\nabla}_{x_k} &= J_{x_k}(x_{k+1})\vec{\nabla}_{x_{k+1}} \end{aligned} \tag{4}$$

$$\begin{aligned} \vec{\nabla}_{x_{k+1}} &= J_{x_{k+1}}(x_{k+2})\vec{\nabla}_{x_{k+2}} \\ \dots \\ \vec{\nabla}_{x_{L-1}} &= J_{x_{L-1}}(x_L)\vec{\nabla}_{x_L} \end{aligned} \tag{5}$$

Obtaining,

$$\vec{\nabla}_{\theta_k} = J_{\theta_k}(x_k)(\prod_{p=k}^{L-1} J_{x_p}(x_{p+1}))\vec{\nabla}_{x_L} \tag{6}$$

where $J_{\text{wrt}}(.)$ are the respective jacobians which can be determined with the layer's expressions and the $\{x_k\}$; and $\vec{\nabla}_{x_L}$ can be determined using $\mathcal{L}$, $\mathcal{R}$ and x_L. This allows to compute the gradients with a complexity that is linear with the number of weights (only one computation of the activations), instead of being quadratic if it were done with the difference quotient expression of the derivatives (one more computation of the activations for each weight).

Backpropagation
We can remark that (5) rewrites as

$$\begin{aligned} \vec{\nabla}_{x_k} &= J_{x_k}(x_{k+1})\vec{\nabla}_{x_{k+1}} \\ &= J_{x'_k}(h(x'_k))J_{x_k}(W_k x_k)\vec{\nabla}_{x_{k+1}} \end{aligned} \tag{7}$$

where $x'_k = W_k x_k$, and these jacobians can be expressed as:

$$\begin{aligned} J_{x'_k}(h(x'_k))[i,j] &= \delta_i^j h'(x'_k[i]) \\ J_{x'_k}(h(x'_k)) &= I\, h'(x'_k) \end{aligned} \tag{8}$$

$$J_{x_k}(W_k x_k) = W_k^T \tag{9}$$

That means that we can write $\vec{\nabla}_{x_k} = (\widetilde{h}_k \circ \widetilde{g}_k)(\vec{\nabla}_{x_{k+1}})$ such that the connectivity matrix $\widetilde{W_k}$ is obtained by transposition. This can be interpreted as gradient calculation being a *back-propagation* on the same neural network, in opposition of the *forward-propagation* done to compute the output.

1.2.4 Some historical advances

Universal approximation

Early researches have shown that neural networks with one level of depth can approximate any real-valued function defined on a compact subset of $\mathbb{R}^n$. This result was first proved for sigmoidal activations (Cybenko, 1989), and then it was shown it did not depend on the sigmoidal activations (Hornik et al., 1989; Hornik, 1991).

For example, this result brings theoretical justification that objective functions exists (even though it does not inform whether an algorithm to approach it exists or is efficient).

Computational difficulty

However, reaching such objective is a computationally difficult problem, which drove back interest from the field. Thanks to better hardware and to using better initialization schemes that speed up learning, researchers started to report more successes with deep neural networks (Hinton et al., 2006; Glorot and Bengio, 2010) ; see (Bengio, 2009) for a review of this period. It ultimately came to a surge of interest in the field after a significant breakthrough on the imagenet dataset (Deng et al., 2009) with deep

CNNs (Krizhevsky et al., 2012). The use of the fast ReLU activation function (Glorot et al., 2011) as well as leveraging graphical processing units with CUDA (Nickolls et al., 2008) were also key factors in overcoming this computational difficulty.

Adoption of ReLU activations
Historically, sigmoidal and tanh activations were mostly used (Cybenko, 1989; LeCun et al., 1989). However in recent practice, the ReLU activation (first introduced as the *positive part*, Jarrett et al., 2009), become the most used activation, as it was demonstrated to be faster and to obtain better results (Glorot et al., 2011). ReLU originated numerous variants *e.g. leaky rectified linear unit* (Maas et al., 2013), *parametric rectified linear unit* (PReLU, He et al., 2015), *exponential linear unit* (ELU, Clevert et al., 2015), *scaled exponential linear unit* (SELU, Klambauer et al., 2017), each one having particular advantages in some applications.

Avoiding overfitting
Neural networks, like any other machine learning technique, may overfit. That is, a model may behave well on the training set but fails to generalize well on unseen examples. The introduction of dropout (Srivastava et al., 2014) have helped models with more parameters to be less prone to overfitting, as dropout consists in hiding some parts of the training samples and their intermediate activations. Another good practice is to normalize per batch (Ioffe and Szegedy, 2015).

Expressivity and expressive efficiency
The study of the *expressivity* (also called *representational power*) of families of neural networks is the field that is interested in the range of functions that can be realized or approximated by this family (Håstad and Goldmann, 1991; Pascanu et al., 2013). In general, given a maximal error ϵ and an objective F, the more expressive is a family $N \subset \mathcal{N}$, the more likely it

is to contain an approximation $f \in N$ such that $d(f, F) < \epsilon$. However, if we consider the approximation $f_{min} \in N$ that have the lowest number of neurons, it is possible that f_{min} is still too large and may be unpractical. For this reason, expressivity is often studied along the related notion of *expressive efficiency* (Delalleau and Bengio, 2011; Cohen et al., 2018).

Rectifier neural netowrks

Of particular interest for the intuition is a result stating that a simply connected neural networks with only ReLU activations (a rectifier neural network) is a piecewise linear function (Pascanu et al., 2013; Montufar et al., 2014), and that conversely any piecewise linear function is also a rectifier neural network such that an upper bound of its depth is logarithmically related to the input dimension (Arora et al., 2018, th. 2.1.). Their expressive efficiency have also been demonstrated compared to neural networks using threshold or sigmoid activations (Pan and Srikumar, 2016).

Benefits of depth

Expressive efficiency analysis have demonstrated the benefits of depth, *i.e.* a shallow neural network would need an unfeasible large number of neurons to approximate the function of a deep neural network (*e.g.* Delalleau and Bengio, 2011; Bianchini and Scarselli, 2014; Poggio et al., 2015; Eldan and Shamir, 2016; Poole et al., 2016; Raghu et al., 2016; Cohen and Shashua, 2016; Mhaskar et al., 2016; Lin et al., 2017; Arora et al., 2018). This field seeks to give theoretical grounds to the practical observation that state-of-the-art architectures are getting deeper.

Benefits of branching operations

Recent works have provided rationales supporting benefits of using branching operations, thus giving justifications for architectures obtained with the generic formalization. In particular, (Cohen et al., 2018) have analyzed the impact of residual connections used in Wavenet-like architec-

tures (Van Den Oord et al., 2016) in terms of expressive efficiency, using tools from the field of tensor analysis ; (Orhan and Pitkow, 2018) have empirically demonstrated that residual connections can resolve some inefficiency problems inherent of fully-connected networks (dead activations, activations that are always equal, linearly dependent sets of activations).

1.2.5 Common layers

Definition 23. Connections
The set of *connections* of a layer (g, h), denoted C_g, is defined as:

$$C_g = \{(i, j), \exists p, W_g[i, j] := \theta_g[p]\}$$

We have $0 \leq |C_g| \leq n_1^{(W_g)} n_2^{(W_g)}$.

Definition 24. Dense layer
A *dense layer* (g, h) is a layer such that $|C_g| = n_1^{(W_g)} n_2^{(W_g)}$, *i.e.* all possible connections exist. The map $(i, j) \mapsto p$ is usually a bijection, meaning that there is no weight sharing.

A neural network made only of dense layers is called a Multi-Layer Perceptron (MLP, Hornik et al., 1989).

Definition 25. Partially connected layer
A *partially connected layer* (g, h) is a layer such that $|C_g| < n_1^{(W_g)} n_2^{(W_g)}$.
A *sparsely connected layer* (g, h) is a layer such that $|C_g| \ll n_1^{(W_g)} n_2^{(W_g)}$.

Definition 26. Convolutional layer
A *n-dimensional convolutional layer* (g, h) is such that the weight kernel θ_g can be reshaped into a tensor w of rank $n + 2$, and such that

$$\begin{cases} I_g \text{ and } O_g \text{ are tensor spaces of rank } n + 1 \\ \forall x \in I_g, g(x) = (g(x)_q = \sum_p x_p *^n w_{p,q})_{\forall q} \end{cases}$$

where p and q index slices along the last ranks.

A neural network that contains convolutional layers is called convolutional neural network (CNN).

Definition 27. Feature maps and input channels
The slices $g(x)_q$ are typically called *feature maps*, and the slices x_p are called *input channels*. Let's denote by $n_o = n_{n+1}^{(O_g)}$ and $n_i = n_{n+1}^{(I_g)}$ the number of feature maps and input channels. In other words, Definition 26 means that for each feature maps, a convolution layer computes n_i convolutions and sums them, computing a total if $n_i \times n_o$ convolutions.

Remark. Note that because they are simply summed, entries of two different input channels that have the same coordinates are assumed to share some sort of relationship. For instance on images, entries of each input channel (typically corresponding to Red, Green and Blue channels) that have the same coordinates share the same pixel location.

Benefits of convolutional layers
Comparatively with dense layers, convolution layers enjoy a significant decrease in the number of weights. For example, an input 2×2 convolution on images with 3-color input channels, would breed only 12 weights per feature maps, independently of the numbers of input neurons. On image datasets, their usage also breeds a significant boost in performance compared with dense layers (Krizhevsky et al., 2012), for they allow to take advantage of the topology of the inputs while dense layers do not (LeCun, Bengio, et al., 1995). A more thorough comparison and explanation of their assets will be discussed in Section 2.1.3.

Decrease of spatial dimensions
Given a tensor input x, the n-dimensional convolutions between the inputs channels x_p and slices of a weight tensor $w_{p,q}$ would result in outputs

y_q of shape $n_1^{(x)} - n_1^{(w)} + 1 \times \ldots \times n_n^{(x)} - n_n^{(w)} + 1$. So, in order to preserve shapes, a padding operation must pad x with $n_1^{(w)} - 1 \times \ldots \times n_n^{(w)} - 1$ zeros beforehand. For example, the padding function of the library *tensorflow* (Abadi et al., 2015) pads each rank with a balanced number of zeros on the left and right indices (except if $n_t^{(w)} - 1$ is odd then there is one more zero on the left).

Definition 28. Padding
A convolutional layer with *padding* (g, h) is such that g can be decomposed as $g = g_{\text{pad}} \circ g'$, where g' is the linear part of a convolution layer as in Definition 26, and g_{pad} is an operation that pads zeros to its inputs such that g preserves tensor shapes.

Remark. One asset of padding operations is that they limit the possible loss of information on the borders of the subsequent convolutions, as well as preventing a decrease in size. Moreover, preserving shape is needed to build some neural network architectures, especially for ones with branching operations *e.g.* examples in Section 1.2.1. On the other hand, they increase memory and computational footprints.

Definition 29. Stride
A convolutional layer with *stride* is a convolutional layer that computes strided convolutions (with stride > 1) instead of convolutions.

Definition 30. Pooling
A layer with *pooling* (g, h) is such that g can be decomposed as $g = g' \circ g_{\text{pool}}$, where g_{pool} is a pooling operation.

Layers with stride or pooling downscale the signals that passes through the layer. These types of layers allows to compute features at a coarser level, giving the intuition that the deeper a layer is in the network, the more abstract is the information captured by the weights of the layer. From a practical point of view, they contribute to lower the computational complexity at deeper levels.

A simple result

In two dimensions, convolutional operations can be rewritten as a matrix-vector multiplication where the matrix is Toeplitz. We show below that it is still the case in n dimensions.

Proposition 31. Connectivity matrix of a convolution with padding

A convolutional layer with padding (g, h) is equivalently defined as its connectivity matrix W_g being a $n_i \times n_o$ block matrix such that its blocks are Toeplitz matrices, and where each block corresponds to a couple (p, q) of input channel p and feature map q.

Proof. Let's consider the slices indexed by p and q, and to simplify the notations, let's drop the subscripts $_{p,q}$. We recall from Definition 12 that

$$
\begin{aligned}
y &= (x *^n w)[j_1, \dots, j_n] \\
&= \sum_{k_1=1}^{n_1^{(w)}} \cdots \sum_{k_n=1}^{n_n^{(w)}} x[j_1 + n_1^{(w)} - k_1, \dots, j_n + n_n^{(w)} - k_n]\, w[k_1, \dots, k_n] \\
&= \sum_{i_1=j_1}^{j_1+n_1^{(w)}-1} \cdots \sum_{i_n=j_n}^{j_n+n_n^{(w)}-1} x[i_1, \dots, i_n]\, w[j_1 + n_1^{(w)} - i_1, \dots, j_n + n_n^{(w)} - i_n] \\
&= \sum_{i_1=1}^{n_1^{(x)}} \cdots \sum_{i_n=1}^{n_n^{(x)}} x[i_1, \dots, i_n]\, \widetilde{w}[i_1, j_1, \dots, i_n, j_n]
\end{aligned}
$$

where $\widetilde{w}[i_1, j_1, \dots, i_n, j_n] =$

$$
\begin{cases}
w[j_1 + n_1^{(w)} - i_1, \dots, j_n + n_n^{(w)} - i_n] & \text{if } \forall t, 0 \leq i_t - j_t \leq n_t^{(w)} - 1 \\
0 & \text{otherwise}
\end{cases}
$$

Using Einstein summation convention as in (2) and permuting indices, we recognize the following tensor contraction

$$
y_{j_1 \cdots j_n} = x_{i_1 \cdots i_n} \widetilde{w}^{i_1 \cdots i_n}{}_{j_1 \cdots j_n} \tag{10}
$$

Following Proposition 11, we reshape (10) as a matrix product. To reshape $y \mapsto Y$, we use the row major order bijections g_j as in (1) defined onto $\{(j_1, \dots, j_n), \forall t, 1 \leq j_t \leq n_t^{(y)}\}$. To reshape $x \mapsto X$, we use the same row major order bijection g_j, however defined on the indices that support non zero-padded values, so that zero-padded values are lost after reshaping. That is, we use a bijection g_i such that $g_i(i_1, i_2, \dots, i_n) = g_j(i_1 - o_1, i_2 - o_2, \dots, i_n - o_n)$ defined if and only if $\forall t, 1 + o_t \leq i_t \leq n_t^{(y)}$, where the $\{o_t\}$ are the starting offsets of the non zero-padded values. $\widetilde{w} \mapsto W$ is reshaped by using g_j for its covariant indices, and g_i for its contravariant indices. The entries lost by using g_i do not matter because they would have been nullified by the resulting matrix product. We remark that W is exactly the block (p, q) of W_g (and not of $W_{g'}$). Now let's prove that it is a Toeplitz matrix.

Thanks to the linearity of the expression (1) of g_j, by denoting $i'_t = i_t - o_t$, we obtain

$$g_i(i_1, i_2, \dots, i_n) - g_j(j_1, j_2, \dots, j_n) = g_j(i'_1 - j_1, i'_2 - j_2, \dots, i'_n - j_n) \tag{11}$$

To simplify the notations, let's drop the arguments of g_i and g_j. By bijectivity of g_j, (11) tells us that $g_i - g_j$ remains constant if and only if $i'_t - j_t$ remains constant for all t. Recall that

$$W[g_i, g_j] = \begin{cases} w[j_1 + n_1^{(w)} - i'_1, \dots, j_n + n_n^{(w)} - i'_n] & \text{if } \forall t, 0 \leq i'_t - j_t \leq n_t^{(w)} - 1 \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

Hence, on a diagonal of W, $g_i - g_j$ remaining constant means that $W[g_i, g_j]$ also remains constants. So W is a Toeplitz matrix.

The converse is also true as we used invertible functions in the index spaces through the proof. □

Remark. Note that the proof does not hold in case there is no padding. This is due to border effects when the index of the n^{th} rank resets in the definition of the row-major ordering function g_j that would be used. Indeed, under appropriate definitions, the matrices could be seen as almost Toeplitz.

This proposition provides an equivalent-characterization of convolutional layers by their connectivity matrix. Therefore, a first avenue to define convolutions on graph signals could be to define them with the connectivity matrix being as in this characterization. However, the Toeplitz property implies that the dimensions have a specific order, which is not possible when dimensions correspond to vertices of a graph. This is because permuting the order of the vertices wouldn't change the graph, but would change the connectivity matrix (which cannot be Toeplitz for every ordering).

1.3 Deep learning on graphs

Deep learning algorithms have been particularly successful for datasets of signals defined on regular domains such as images or time series. One key ingredient of their success is the use of convolutions. However, classical convolutions are only defined on Euclidean domains, so that extending deep learning on non-Euclidean domains is not straightforward. One way to represent a non-Euclidean domain is through a graph *i.e.* as points (the vertices) and relations between them (the edges). This field also comprises the study of deep learning on manifolds (*i.e.* locally Euclidean domains). The field of deep learning on graphs or manifolds have been called recently Geometric Deep Learning (Bronstein et al., 2017). In this manuscript, we are only interested in deep learning on graphs.

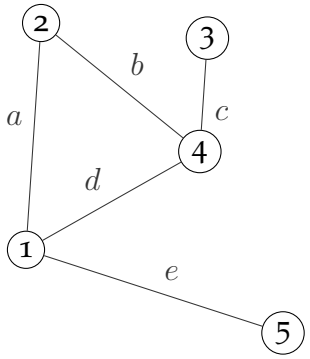
1.3.1 Graph and signals

We present the vocabulary, notation and conventions we will employ for graphs and signals.

Definition 32. Graph
A *graph* G is a couple of countable vertex and edge sets $\langle V, E \rangle$ *s.t.* $E \subset V^2$.

The terms *vertex* and *node* are used interchangeably. Additionaly, we consider that a graph is always *simple i.e.* no two edges share the same set of vertices. Unless stated otherwise, a graph is undirected, *i.e.* (u, v) and (v, u) refer to the same edge. When it is not the case, it is called a *digraph*. We define the relation $u \sim v \Leftrightarrow (u, v) \in E$. We precise the graph if needed over the symbol $\overset{G}{\sim}$. For a digraph we use the symbol $\rightarrow$ instead of $\sim$. A *path* is a sequence $v_1 \sim \cdots \sim v_r$. It is said to be *simple* if its vertices are distincts, except possibly for the first and last. A graph is said to be *connected* if there exists a path from any vertex to any other vertex. We define the *neighborhood* of a vertex as $\mathcal{N}_u = \{v \in V, u \sim v\}$. For digraphs, it is equal

to the union of the *in-* and *out*-neighborhoods. We only consider graphs without isolated vertex (a vertex with an empty neighborhood). We also only consider *weighted* graphs. That is, a graph $G = \langle V, E \rangle$ is associated with a weight mapping $w : V^2 \to \mathbb{R}+$ *s.t.* $w(u, v) = 0 \Leftrightarrow u \nsim v$. If G is finite, its *adjacency matrix* $A \in \mathbb{R}^{V \times V}$ is defined *w.r.t.* to a vertex ordering $V = \{v_1, \ldots, v_n\}$ as $A[i, j] = w(v_i, v_j)$. Figure 7 illustrates an example of a graph and its adjacency matrix.



$$\begin{pmatrix} 0 & a & 0 & d & e \\ a & 0 & 0 & b & 0 \\ 0 & 0 & 0 & c & 0 \\ d & b & c & 0 & 0 \\ e & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 7: Example of a graph and its adjacency matrix

The *order* of G is equal to its number of vertices, possibly infinite. The *degree* of a vertex v is equal to the number of edges it is attached to. For digraphs the degree is the sum of the *in-* and *out*-degrees. The *degree* of G refers to its max degree. G is said to be *degree-regular* if all its vertices have the same degree. If it is finite, its *degree matrix* D (*w.r.t.* to a vertex ordering $V = \{v_1, \ldots, v_n\}$) is the diagonal matrix for which the diagonal entry corresponding to a vertex is the sum of the weights of the edges it is part of. Its *laplacian matrix* L is the substraction $L = D - A$, which can be *normalized* $L = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$, *left-normalized* $L = I - D^{-1}A$, or *right-normalized* $L = I - AD^{-1}$. The same naming convention is used for normalized version of the adjacency matrix A. A subgraph of G induced by a subset $U \subset V$ is the graph with vertex and edge set restricted by U. The *complement* graph G^C shares the same vertex set but $u \overset{G^C}{\sim} v \Leftrightarrow u \overset{G}{\nsim} v$. A

complete graph is such that there exists an edge between any two vertices.

Definition 33. Grid graph
Let a graph $G = \langle V, E \rangle$ such that the expression $u \sim v \Leftrightarrow \|u - v\|_1 = 1$ makes sense. G can be called:

- a *grid graph* if $V = \mathbb{Z}^2$
- a *finite grid graph* if $\exists (n, m) \in \mathbb{Z}^2, V = [\![1, n]\!] \times [\![1, m]\!]$
- a *circulant grid graph* if $\exists (n, m) \in \mathbb{Z}^2, V = \mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}$

Definition 34. Bipartite graph
A graph is called *bipartite* if its vertex set is a disjoint union $V = V_1 \cup V_2$ *s.t.*

$$u \sim v \Rightarrow (u, v) \in V_1 \times V_2 \vee (u, v) \in V_2 \times V_1$$

If it is finite, its *bipartite-adjacency* matrix $A \in \mathbb{R}^{V_1 \times V_2}$ is a rectangular matrix defined *w.r.t.* to a vertex ordering $V_1 = \{u_1, \ldots, u_n\}$, $V_2 = \{v_1, \ldots, v_n\}$ and weight mapping w as $A[i, j] = w(u_i, v_j)$.

Definition 35. Signal
A *signal* on V, $s \in \mathcal{S}(V)$, is a function $s : V \to \mathbb{R}$. The *signal space* $\mathcal{S}(V)$ is the linear space of signals on V.

Remark. In particular, a vector space, and more generally a tensor space, are finite-dimensional signal spaces on any of their bases. Reciprocally, a signal space is a linear space which canonical basis is the signal space domain. Therefore, signals can be represented as vectors (or tensors) and then fed to neural networks.

A *graph signal* on a graph $G = \langle V, E \rangle$ is a signal on its vertex set V. We denote by $\mathcal{S}(G)$ or $\mathcal{S}(V)$ the graph signal space. G can be referred as the *underlying structure* of $\mathcal{S}(V)$. An *entry* of a signal s is an image by s of some $v \in V$ and we denote $s[v]$. If v is represented by an n-tuple, we can also write $s[v_1, v_2, \ldots, v_n]$. The *support* of a signal $s \in \mathcal{S}(V)$ is the subset

$\mathrm{supp}(s) \subset V$ on which $s \neq 0$. For spaces of signals that aren't real-valued, their codomain $\mathbb{E}$ is precised in the subscript $\mathcal{S}_{\mathbb{E}}(V)$. The signal Id is the identity function. Given $v \in V$, the dirac signal $\delta_v \in \mathcal{S}(V)$ is the signal valued as 1 on v and 0 everywhere else. The family of all dirac signals spans $\mathcal{S}(V)$.

1.3.2 Learning tasks

There are two main tasks related to deep learning and graph signals.

Supervised classification of graph signals

This is the classical application of deep learning transposed to graph signals, rather than image or audio signals. It is the main task we will have in mind in the course of this manuscript. Given a graph $G = \langle V, E \rangle$ and an input signal $x \in \mathcal{S}(G)$ the goal is to classify x. If there are c possible classes, a neural network f outputs a vector $y = f(x)$ of dimension c, and its dimension with the biggest weight determines the predicted class. Indeed, a standard MLP can be trained on a dataset of graph signals. However, an MLP wouldn't take the graph structure G into consideration. By similarity with CNNs that leverage the grid structure of images to achieve better performances than MLPs, a challenge is to define a neural network on graph signals that can leverage G. We review some models from the literature in Section 1.3.4 and in Section 1.3.5. We develop an algebraic understanding in Chapter 2 of why and how they should work, and also propose our own models and point of view in Chapter 3.

Semi-supervised classification of nodes

This task is in some way obtained from a transposed perspective of the previous one. Given a dataset of graph signals, represented as a matrix $X \in \mathbb{R}^{n \times N}$, where the rows represent the nodes, and the columns represent the signals, the goal is to classify the nodes. This amounts to classify

the rows, whereas the previous task amounts to classify the columns. As opposed to the previous one, this task is *transductive i.e.* every node data is available during training, including those from the validation and test set (but their labels are not), and it is *semi-supervised i.e.* some nodes have no label. This allows to learn on much more data than if we were restricted to labeled data. In this task, the edges connect learning samples, however in the previous one, the edges were connecting features of learning samples. This is this edge relationship between learning samples that renders the semi-supervised approach possible. This task have received much more attention than the previous one in the recent literature. We explain why in Section 1.3.4.

Other learning tasks

In this manuscipt, we are less interested in other deep learning tasks related to graphs, so we briefly discuss them here. One is supervised classification of graphs, which is different than classifying graph signals. Examples include (Niepert et al., 2016; Tixier et al., 2017; Nikolentzos et al., 2017; Bai et al., 2018). Another related interesting task is called representation learning of nodes, which tackles the challenge to learn a linear representation of nodes. A common approach, derived from word2vec (Mikolov et al., 2013b; Mikolov et al., 2013a), is called node2vec (Grover and Leskovec, 2016), and was later improved in graphSAGE (Hamilton et al., 2017a). A review on this subject is done by Hamilton et al., 2017b. A thorough survey on representation learning for networks is given in (Zhang et al., 2017).

1.3.3 Datasets

We present here the standard datasets that we will use in this manuscript.

Images

Images are signals on grid graphs. Therefore they constitute a first test for models that are designed to be able to classify graph signals. A second step is to test on scrambled versions of image datasets, *i.e.* a random permutation of the pixels is fixed and the tested models are fed with images whose pixels have been shuffled according to this permutation. Therefore the domains of the scrambled input signals are not grid graphs.

- MNIST (LeCun et al., 1998) is a dataset of handwritten digits of size 28x28. It contains 10 classes and is splitted between 50'000 samples for training and 10'000 samples for testing.
- CIFAR-10 (Krizhevsky, 2009) is a dataset of tiny pictures of size 32x32. It contains 10 classes and is splitted between 50'000 samples for traning and 10'000 samples for testing.
- Scrambled MNIST is the scrambled version of MNIST. A graph based on nearest neighbours from the pixel covariances is used to represent each scrambled sample as a graph signal.
- Scrambled CIFAR-10 is analoguous.

Functional magnetic resonance imagings (fMRI)

fMRI samples can be represented by graph signals. The graphs are resembling grid graphs to some extent since they are embedded in an Euclidean space.

- The PINES dataset consists of fMRI scans on 182 subjects, during an emotional picture rating task (Chang et al., 2015). In (Lassance et al.,

2018), we fetched individual first-level statistical maps (beta images) for the minimal and maximal ratings from `https://neurovault.org/collections/1964/`, to generate the dataset. Full brain data was masked on the MNI template and resampled to a 16mm cubic grid, in order to reduce dimensionality of the dataset while keeping a regular geometrical structure to infer the graph. Final volumes used for classification contain 369 signals for each subject and rating.

Text documents

Text documents can be represented as graph signals. Words are the vertices, and the value of a signal at a given vertex corresponds to a normalized occurence of the corresponding word. Edges connect nearest neighbours in some metric space.

- 20NEWS (Joachims, 1996) is a dataset of text documents. It contains 20 classes and is splitted between 11'314 samples for training and 7'532 samples for testing. Each document is represented by a bag-of-word. In the version used by Defferrard et al., 2016, that we consider, documents are treated as signals on a graph of 10'000 vertices which represent the 10'000 most common words (the other words are not used). Edges are drawn from each vertex to their 16 nearest neighbours in the cosine similarity metric space.

Citation networks

In a citation network (*i.e.* a graph of citations), nodes are bag-of-word documents and edges represent citations. Models are tested on a citation network to the task of semi-supervised classification of the nodes. We use three standard datasets: Cora, Citeseer and Pubmed (Sen et al., 2008), for which we follow the experimental settings of Yang et al., 2016, and the dataset split from Kipf and Welling, 2016. In particular, there are only 20 training samples per class, but they are connected to other samples.

- Cora is a dataset of 2'708 nodes of dimension 1'433 with 5'429 edges. It contains 7 classes and consists of 140 samples for training, 500 samples for validation, 1'000 samples for testing, and 1'068 unlabelled samples.

- Citeseer is a dataset of 3'327 nodes of dimension 3'703 with 4'732 edges. It contains 6 classes and consists of 120 samples for training, 500 samples for validation, 1000 samples for testing, and 1'707 unlabelled samples.

- Pubmed is a dataset of 19'717 nodes of dimension 500 with 44'338 edges. It contains 3 classes and consists of 60 samples for training, 500 samples for validation, 1000 samples for testing, and 18'157 unlabelled samples.

1.3.4 Spectral methods

Spectral methods are based on spectral graph theory (Chung, 1996) which aims at characterizing structral properties of a graph $G = \langle V, E \rangle$ through the eigenvalues of the laplacian matrix L. In particular, since it is hermitian, it admits a complete set of normalized eigenvectors. By fixing a normalized eigenvector basis ordered in the rows of U (by ascending eigenvalues), U is used to define the *Graph Fourier Transform* (GFT) of a signal $s \in \mathcal{S}(G)$ (Shuman et al., 2013), and the conjugate-transpose U^* defines the inverse GFT. We write

$$\widehat{s} = Us \tag{13}$$

$$\widetilde{s} = U^* s \tag{14}$$

Remark. The GFT extends the notion of *Discrete Fourier Transform* (DFT) to general graphs, since that for circulant grid graphs U can be the DFT matrix.

By analogy with the convolution theorem, a convolution can be defined as pointwise multiplication, denoted $\cdot$, in the spectral domain of the graph (Hammond et al., 2011). For $s, g \in \mathcal{S}(G)$, we have:

$$s * g = \widehat{\widehat{s} \cdot \widehat{g}} \tag{15}$$

This expression can be used to define convolutional layers and spectral CNNs on graphs. However, Bruna et al., 2013 pointed out that (15) would generate filters with $\mathcal{O}(n)$ weights, where n is the order of G. So they proposed to learn filters θ with only $\mathcal{O}(1)$ weights and then to smoothly interpolate the remaining weights as $g = K\theta$, where K is a linear smoother matrix. They motivate their construction by the fact that smooth multipliers in the spectral domain should simulate local operations in the vertex domain. To elaborate a bit on this, note that we have:

$$Ls[u] = \sum_{v \in V} w(u, v)(s[u] - s[v]) \tag{16}$$

And so,

$$\begin{aligned} s^T Ls &= \sum_{u \in V} \sum_{v \in V} w(u, v) s[u](s[u] - s[v]) \\ &= \frac{1}{2} \sum_{u \in V} \sum_{v \in V} w(u, v) s[u](s[u] - s[v]) + \frac{1}{2} \sum_{v \in V} \sum_{u \in V} w(v, u) s[v](s[v] - s[u]) \\ &= \sum_{u \in V} \sum_{v \in V} \frac{w(u, v)}{2} (s[u] - s[v])^2 \end{aligned} \tag{17}$$

That is, $s^T Ls$ is some sort of measure of *smoothness* of the signal s, penalized by the weights w. The bigger is $w(u, v)$, the closest $s(u)$ and $s(v)$ must

be to lower the smoothness (17). Since L is symmetric, its eigenvalues are non-negative real numbers, and U diagonalizes L as $\Lambda = ULU^*$. Denote $(\lambda_i)_i$ the eigenvalues, the smoothness measure rewrites:

$$s^T L s = \widehat{s}^* \Lambda \widehat{s} = \sum_{i=1}^{n} \lambda_i \widehat{s}[i]^2 \tag{18}$$

Therefore, as they pointed out, smoothness of s can be read off the coordinates of $\hat{s}$, like for the DFT. Moreover, spectral multipliers modulate its smoothness, and decay in the spectral domain is related to smoothness in the vertex domain. But contrary to their conjecture, smoothness in the spectral domain is not necessary related to decay is the vertex domain (and so to some form of locality). For instance, since the laplacian L^C of the complement graph G^C commutes with L, it can share the same eigenvector basis U, and thus define the same GFT, but their notion of locality in the vertex domain are opposed. Another drawback is that this method requires computing the GFT which complexity is at least $\mathcal{O}(n^2)$ as there is no equivalent of the Fast Fourier Transform (FFT) on graphs, so the authors suggest to use a lower number of eigenvectors $d < n$ from the laplacian eigenbasis.

Then, Defferrard et al., 2016 remedy to these issues by proposing an approximate formulation based on the Chebychev polynomials, denoted by $(T_i)_i$, where i is the polynomial order. That is, their proposed approximate filters are in the form

$$g_\theta(L) = \sum_{i=0}^{k} \theta[i]\, T_i(\widetilde{L}) \tag{19}$$

where $\widetilde{L} = \frac{\lambda_{\max}}{2} L - I_n$ is the scaled normalized laplacian with eigenvalues lying in the range $[-1, 1]$. $g_\theta(L)$ are spectral multipliers since we have:

$$g_\theta(L)s = g_\theta(U^*\Lambda U)s = U^* g_\theta(\Lambda) U s$$

$$= \widetilde{g_\theta(\Lambda)\mathbf{1}} * s \tag{20}$$

These filters enjoy locality properties, they contain $\mathcal{O}(1)$ weights, and their complexity is $\mathcal{O}(n)$ when rows of L are sparse. The use of truncated Chebychev expansion (Hammond et al., 2011) ensures that in theory any set of spectral multipliers can be approximated. Also, since they are laplacian polynomials, some authors would argue that these filters are transferable from one graph to another. From a combinatorial point of view this is true. However there is no reason that spectral multipliers from a spectral domain make sense in another one, and there are no experiment in the literature to support the hypothesis. On the other hand, (Yi et al., 2016) (who do not use polynomial filters) fix a canonical spectral base in order to synchronize every spectral domains. Their idea is to learn a warping from any eigenbasis to the canonical one, prior to performing spectral multiplication, in the manner of spatial transformer networks (STN, Jaderberg et al., 2015).

However, it is hard to evaluate if a model performs well on the task of supervised classification of graph signals, because there are not much known datasets in the literature for which the given graph domain holds enough information.

For example, Defferrard et al. built a graph signal dataset from the text categorization dataset 20NEWS (Joachims, 1996, see Section 1.3.3). However, their model (ChebNet32) fails to surpass Multinomial Naive Bayes (MNB). Moreover, even though they report that their model beats MLPs, but through experiments we noticed the contrary. In results we report in Table 1, we see that a lighter MLP, composed of a single Fully-Connected (FC) layer with ReLU and 20% dropout outperforms ChebNet32. We replicated their preprocessing phase from the code on their official repository and averaged our results on 100 runs of 10 epochs[1].

[1] A few epochs are enough since models seem to overfit fast on this dataset.

MNB	FC2500	FC2500-FC500	ChebNet32	FC500
68.51^a	64.64^a	65.76^a	68.26^a	$\mathbf{71.96} \pm 0.15^b$

[a] As reported in Defferrard et al., 2016
[b] From our experiments.

Table 1: Accuracies (in %) on 20NEWS

Despite the significant theoretical contribution, this negative result stresses out the importance of the graph used in practice to support the convolution, a point that they also discussed. Henaff et al., 2015, proposed supervised graph estimation techniques, but a better graph signal dataset would be one that come with an already suitable graph, that of current literature is still lacking.

On the other hand, attention in the domain has shifted toward the task of semi-supervised classification of nodes, where good datasets are not lacking. For example, Levie et al., 2017, mainly demonstrate the usefulness of their model on these type of tasks. They define polynomial filters, for which Chebychev filters are a special case, that are capable to specialize in narrow bands of frequency in the spectral domain.

Another spectral avenue consists in using wavelets defined in the graph spectral domain (Hammond et al., 2011), in order to build a scattering network (Bruna and Mallat, 2013; Chen et al., 2014). This idea have been exploited recently by Zou and Lerman, 2018, then by Gama et al., 2018.

1.3.5 Vertex-domain methods

As their name suggests, vertex-domain methods operates directly on the vertices of the graph. Convolution can be modelized as a function f of the kernel weights θ and neighboring vertices (contained in the local receptive field $\mathcal{R}(v)$), usually based on dot products. That is

$$y[v] = f_\theta \left(\{ u \in \mathcal{R}(v) \} \right) \tag{21}$$

As such, it retains the property of being localized and of sharing weights in some way. But there remains the need to specify how the shared weights are allocated in this local receptive field (Vialatte et al., 2016). This allocation can depend on *e.g.* an arbitrary order (Niepert et al., 2016), on a diffusion process (Atwood and Towsley, 2016), on a function of both vertices and their neighbors (Monti et al., 2016; Simonovsky and Komodakis, 2017), on a random walk (Hechtlinger et al., 2017), on another learned kernel (Vialatte et al., 2017), on an attention mechanism (Velickovic et al., 2017; Lee et al., 2018), on pattern identification (Sankar et al., 2017), or on translation identification (Pasdeloup et al., 2017a). All these methods differ in the function f, but in the end, their definition highly overlap. That is why some authors have proposed unified frameworks (Gilmer et al., 2017).

In particular, Kipf and Welling, 2016, were first to transpose ChebNet to the task of semi-supervised node classification. Chebychev filters (19) then take a form that is interpretable in the vertex domain, which is

$$Y = \sum_{i=0}^{k} T_i(\widetilde{L}) X \Theta \tag{22}$$

where $X \in \mathbb{R}^{n \times N}$, $\Theta \in \mathbb{R}^{N \times M}$, n is the number of nodes, N is the number of input channels (features per node), and M is the number of output

feature maps. On the left, powers of $\widetilde{L}$ diffuse the graph signal X to share node information. On the right, Θ maps the diffused signals to another representation. So in essence, this formulation is more a vertex-domain method. They found that the best performing filters were expressed in a simplified form

$$Y = \widetilde{A}X\Theta \tag{23}$$

where $\widetilde{A}$ is the normalized adjacency matrix of the graph to which self-loops are added. They call the architecture composed with these filters a Graph Convolution Network (GCN). Similiarly, $\widetilde{A}X$ shares node information via the edges and Θ makes the model learns. This fomulation attracted a lot of research attention and was, in particular, extended with attention mechanism (no pun intended), inspired from the field of neural machine translation (Bahdanau et al., 2014). A review is done by Lee et al., 2018.

For example, Velickovic et al., 2017, propose a model that learns attention in a local receptive field. They call it Graph ATtention network (GAT). The attention mechanism is parameterized by a neural network a, containing a single FC layer $(g, \text{LeakyReLU})$, which takes as input a couple of neighboring nodes (i, j) and outputs a scalar $\alpha_{i,j}$. The attention that i deserves to j is:

$$\alpha_{i,j} = \text{softmax}_j \left(a(X[i,:]\,\Theta \parallel X[j,:]\,\Theta) \right) \tag{24}$$

where $\parallel$ denotes concatenation. In a sense, a learns which input feature maps are most useful to describe the attention a node should derserve to another. The forward propagation is done similarly than (23), but with a matrix A_k filled with the attention coefficients $\alpha_{i,j}$ instead of $\widetilde{A}$. They also propose that the model learns multiple attention heads, so that a GAT

layer amounts to:

$$Y = \Big\Vert_{k=1}^{K} A_k X \Theta_k \tag{25}$$

$$\text{or } Y = \frac{1}{K} \sum_{k=1}^{K} A_k X \Theta_k \tag{26}$$

Another method called Topology Adaptative GCN (TAGCN, Du et al., 2017) uses a convolution filter borrowed from graph signal processing literature (Sandryhaila and Moura, 2013), which is defined in the vertex domain as:

$$Y[:, f] = \sum_{c=1}^{N} \left(\sum_{k=1}^{K} \Theta_k[c, f] \, \widetilde{A}^k \right) X[:, c] \tag{27}$$

It can be rewritten as:

$$Y = \sum_{k=1}^{K} \widetilde{A}^k X \Theta_k \tag{28}$$

Each succesive powers of the normalized adjacency matrix $\widetilde{A}$ allows for considering wider neighborhoods.
Other works extending or resembling GCN are numerous in recent days (*e.g.* Niepert and Garcia-Duran, 2018). We do not cover them since that their novelty compared to GCN is limited.

In the next chapter, we study how to characterize a convolution of graph signals in the vertex domain.

Chapter 2
Convolution of graph signals

Contents

Chapter overview

Defining a convolution of signals over graph domains is a challenging problem. If the graph is not a grid graph, there exists no natural extension of the Euclidean convolution. In Section 2.1, we analyze the reasons why the Euclidean convolution operator is useful in deep learning. In particular, we recall a classical characterization: that convolution operators are exactly the class of linear functions that are equivariant to translations (Theorem 41). Therefore, we then search for domains onto which a convolution with these properties can be naturally obtained. This leads us to put our interest on representation theory and convolutions defined on groups. Since the Euclidean convolution is just a particular case of the group convolution, it makes perfect sense to steer our construction in this direction. In Section 2.2, we seek to transfer the definition of the group convolution onto the vertex domain, through its symmetric group. We do not obtain the wanted characterization with a simple homomorphism condition (Proposition **??**). However, we manage to obtain it should we fix an equivariant mapping between a subgroup of the symmetric group and the vertex domain (Theorem 54). Then, we propose a mixed formulation of this convolution as a binary operation between a signal defined on the vertex domain and a signal defined on its corresponding subgroup, for which we demonstrate that the characterization holds under abelianity (Theorem 61). In Section 2.3, we introduce the role of the edge set and see how it influences the construction. In particular, we obtain a characterization of graphs that admit a natural construction by Cayley subgraph isomorphism (Theorem 64). We analyze the notions of locality and weight sharing in this construction, and give a formulation for small kernels. Then, in Section 2.4, we relax some aspect of the construction to adapt it to general graphs. We explain why a construction based on groups fails for some graphs, and introduce the notion of groupoid. We extend

the previous construction with groupoids of partial transformations, and prove that under a mild condition the characterization is preserved (Theorem 82). Finally, we extend it another time with another type of groupoid, that we call path groupoids. Path groupoids allows to tackle the more general case, and for them we obtain the characterization should we fix a way to traverse the vertex set using a subset of the edge set (Theorem 89). We summarize our constructions in a conclusive Section 2.5. The result of this chapter is the obtention of a set of general expressions and theorems that describe convolutions of graph signals.

2.1 Analysis of the classical convolution

In this section, we are exposing a few properties of the classical convolution that a generalization to graphs would likely try to preserve. For now let's consider a graph G agnostically of its edges *i.e.* $G \cong V$ is just the set of its vertices.

2.1.1 Properties of the convolution

Consider an edge-less grid graph *i.e.* $G \cong \mathbb{Z}^2$. By restriction to compactly supported signals, this case encompass the case of images.

Definition 36. Convolution on $\mathcal{S}(\mathbb{Z}^2)$
Recall that the (discrete) convolution between two signals s_1 and s_2 over $\mathbb{Z}^2$ is a binary operation in $\mathcal{S}(\mathbb{Z}^2)$ defined as:

$$\forall (a,b) \in \mathbb{Z}^2, (s_1 * s_2)[a,b] = \sum_i \sum_j s_1[i,j]\, s_2[a-i, b-j]$$

Definition 37. Convolution operator
A *convolution operator* is a function of the form $f_w : x \mapsto x * w$, where x and w are signals of domains for which the convolution $*$ is defined. When $*$ is not commutative, we differentiate the *right* operator $x \mapsto x * w$ from the *left* one $x \mapsto w * x$.

The following properties of the convolution on $\mathbb{Z}^2$ are of particular interest for our study.

Linearity
Operators produced by the convolution are linear. So they can be used as linear parts of layers of neural networks.

Locality and weight sharing

When w is compactly supported on K, an impulse response $f_w(x)[a, b]$ amounts to a weighted aggregation of entries of x in a neighborhood of (a, b), called the *Local Receptive Field* (LRF). The weight kernel w used for the aggregation is fixed *w.r.t.* (a, b), so that we say that the weights are shared.

Commutativity

This convolution is commutative. However, it won't necessarily be the case on other domains.

Equivariance to translations

Convolution operators are equivariant to translations. Below, we show that the converse of this result also holds with Theorem 41.

2.1.2 Characterization on grid graphs

We first define what a transformation of a domain is, and how it can be extended to signals defined on this domain.

Definition 38. Transformation

A *transformation* $f : V \to V$ is a function with same domain and codomain. The set of transformations is denoted $\Phi(V)$. The set of invertible transformations is denoted $\Phi^*(V) \subset \Phi(V)$. If V is a linear space, the set of its linear transformation is denoted $\mathcal{L}(V)$.

Remark. Note that $\Phi^*(V)$ forms what is usually called the symmetric group of V.

$\Phi^*(V)$ can *move* signals of $\mathcal{S}(V)$ by linear extension of its group action, as we explain with the lemma that follows.

Lemma 39. Extension of $\Phi^*(V)$ to $\mathcal{L}(\mathcal{S}(V))$
An invertible transformation $f \in \Phi^*(V)$ can be extended linearly to the signal space $\mathcal{S}(V)$, and we have:

$$\forall s \in \mathcal{S}(V), \forall v \in V, f(s)[v] = s[f^{-1}(v)]$$

Proof. Let $s \in \mathcal{S}(V), f \in \Phi^*(V), L_f \in \mathcal{L}(\mathcal{S}(V))$ *s.t.* $\forall v \in V, L_f(\delta_v) = \delta_{f(v)}$. Then, by linear decomposition of s on the family of dirac signals, we have:

$$\begin{aligned} L_f(s) &= \sum_{v \in V} s[v]\, L_f(\delta_v) \\ &= \sum_{v \in V} s[v]\, \delta_{f(v)} \\ \text{So, } \forall v \in V, L_f(s)[v] &= s[f^{-1}(v)] \end{aligned}$$

L_f extends f to $\mathcal{S}(V)$. When there is no ambiguity, we use the same symbol for $f(.)$ and $L_f(.)$. □

For translations, we use the following formalism.

Definition 40. Translation on $\mathcal{S}(\mathbb{Z}^2)$
A translation on $\mathbb{Z}^2$ is defined as a transformation $t \in \Phi^*(\mathbb{Z}^2)$ such that

$$\exists (a, b) \in \mathbb{Z}^2, \forall (x, y) \in \mathbb{Z}^2, t(x, y) = (x + a, y + b)$$

By Lemma 39, it also acts on $\mathcal{S}(\mathbb{Z}^2)$ with the notation $t_{a,b}$ *i.e.*

$$\forall s \in \mathcal{S}(\mathbb{Z}^2), \forall (x, y) \in \mathbb{Z}^2, t_{a,b}(s)[x, y] = s[x - a, y - b]$$

The next theorem fully characterizes convolution operators with their translational equivariance property. This can be seen as a discretization of a classic result from the theory of distributions (Schwartz, 1957).

Theorem 41. Characterization of convolution operators on $\mathcal{S}(\mathbb{Z}^2)$
On real-valued signals over $\mathbb{Z}^2$, the class of linear transformations that are equivariant to translations is exactly the class of convolutive operations *i.e.*

$$\exists w \in \mathcal{S}(\mathbb{Z}^2), f = . * w \Leftrightarrow \begin{cases} f \in \mathcal{L}(\mathcal{S}(\mathbb{Z}^2)) \\ \forall t \in \mathcal{T}(\mathcal{S}(\mathbb{Z}^2)), f \circ t = t \circ f \end{cases}$$

Proof. The result from left to right is a direct consequence of the definitions:

$$\begin{aligned}
&\forall s \in \mathcal{S}(\mathbb{Z}^2), \forall s' \in \mathcal{S}(\mathbb{Z}^2), \forall (\alpha, \beta) \in \mathbb{R}^2, \forall (a, b) \in \mathbb{Z}^2, \\
&\qquad f_w(\alpha s + \beta s')[a, b] = \sum_i \sum_j (\alpha s + \beta s')[i, j]\, w[a - i, b - j] \\
&\qquad\qquad = \alpha f_w(s)[a, b] + \beta f_w(s')[a, b] \qquad \text{(linearity)} \\
&\forall s \in \mathcal{S}(\mathbb{Z}^2), \forall (x, y) \in \mathbb{Z}^2, \forall (a, b) \in \mathbb{Z}^2, \\
&\qquad f_w \circ t_{x,y}(s)[a, b] = \sum_i \sum_j t_{x,y}(s)[i, j]\, w[a - i, b - j] \\
&\qquad\qquad = \sum_i \sum_j s[i - x, j - y]\, w[a - i, b - j] \\
&\qquad\qquad = \sum_{i'} \sum_{j'} s[i', j']\, w[a - i' - x, b - j' - y] \qquad (29) \\
&\qquad\qquad = f_w(s)[a - x, b - y] \\
&\qquad\qquad = t_{x,y} \circ f_w(s)[a, b] \qquad \text{(equivariance)}
\end{aligned}$$

Now let's prove the result from right to left.

Let $f \in \mathcal{L}(\mathcal{S}(\mathbb{Z}^2))$, $s \in \mathcal{S}(\mathbb{Z}^2)$ and we suppose that f commutes with translations. By linear decomposition of s on the family of dirac signals,

we have:

$$\begin{aligned} f(s) &= \sum_i \sum_j s[i,j]\, f(\delta_{i,j}) \\ &= \sum_i \sum_j s[i,j]\, f \circ t_{i,j}(\delta_{0,0}) \\ &= \sum_i \sum_j s[i,j]\, t_{i,j} \circ f(\delta_{0,0}) \end{aligned}$$

By denoting $w = f(\delta_{0,0}) \in \mathcal{S}(\mathbb{Z}^2)$, we obtain:

$$\begin{aligned} \forall (a,b) \in \mathbb{Z}^2, f(s)[a,b] &= \sum_i \sum_j s[i,j]\, t_{i,j}(w)[a,b] \\ &= \sum_i \sum_j s[i,j]\, w[a-i, b-j] \\ \textit{i.e. } f(s) &= s * w \end{aligned} \tag{30}$$

□

2.1.3 Usefulness of convolutions in deep learning

Equivariance property of CNNs

In deep learning, an important argument in favor of CNNs is that convolutional layers are equivariant to translations. Intuitively, that means that a detail of an object in an image should produce the same features independently of its position in the image.

Lossless superiority of CNNs over MLPs

The converse result, as a consequence of Theorem 41, is never mentioned in deep learning literature. However it is also a strong one. For example, let's consider a linear function that is equivariant to translations. Thanks to the converse result, we know that this function is a convolution operator parameterized by a weight vector w, $f_w : . * w$. If the domain is

compactly supported, as in the case of images, we can break down the information of w in a finite number n_q of kernels w_q with small compact supports of same size (for instance of size 2×2), such that we have $f_w = \sum_{q \in \{1,2,\dots,n_q\}} f_{w_q}$. The convolution operators f_{w_q} are all in the search space of 2×2 convolutional layers. In other words, every translational equivariant linear function can have its information parameterized by these layers. So that means that the reduction of parameters from an MLP to a CNN is done without loss of expressivity (provided the objective function is known to bear this property). Besides, it also helps the training to search in a much more confined space. For example, on CIFAR-10 (see description in Section 1.3.3), CNNs reportedly attain up to 2.31% error on classification (Yamada et al., 2018), while MLPs plateaued at 21.38% (Lin et al., 2015). Intuitively, the reason for this success is *simplification by symmetry*: the supposed translational equivariance of the objective function is a symmetry that is exploited by the convolution layer to simplify its input.

Methodology for extending to general graphs

Hence, in our construction, we will try to preserve the characterization from Theorem 41 since it is mostly the reason why they are successful in deep learning. Note that other useful properties are also a consequence of this characterization. For example, the fact that convolutional layers have less parameters than a dense layer is also a consequence of this characterization.

2.2 Construction on the vertex set

As Theorem 41 is a complete characterization of convolutions, it can be used to define them *i.e.* convolution operators can be constructed as the set of linear transformations that are equivariant to translations. However, in the general case where G is not a grid graph, translations are not defined, so that construction needs to be generalized beyond translational equivariances.

In mathematics, convolutions are more generally defined for functions defined over a group structure. The classical convolution that is used in deep learning is just a narrow case where the domain group is an Euclidean space. Therefore, constructing a convolution on graphs should start from the more general definition of convolution on groups rather than convolution on Euclidean domains.

Our construction is motivated by the following questions:

- Does the equivariance property holds ? Does the characterization from Theorem 41 still holds ?
- Is it possible to extend the construction on non-group domains, or at least on mixed domains ? (*i.e.* one signal is defined over a set, and the other is defined over its transformations).
- Can a group domain draw an underlying graph structure ? Is the group convolution naturally defined on this class of graphs ?
- Can we characterize graphs accepting our construction ?

In this section, we first aim at transferring the group convolution onto the vertex set. Then, in Section 2.3, we will see the implications of considering the edge set in the process.

2.2.1 Preliminaries

We first recall preliminary notions about groups and group convolutions.

Definition 42. Group
A group Γ is a set equipped with a closed, associative and invertible composition law that admits a unique left-right identity element e.

When the law is commutative, the group is said to be *abelian*. The group convolution extends the notion of the classical discrete convolution and is defined as follows.

Definition 43. Group convolution
Let a group Γ, the *group convolution* between two signals s_1 and $s_2 \in \mathcal{S}(\Gamma)$ is defined as:

$$\forall h \in \Gamma, (s_1 *_\Gamma s_2)[h] = \sum_{g \in \Gamma} s_1[g]\, s_2[g^{-1}h]$$

provided at least one of the signals has finite support if Γ is not finite.

We call *left multiplication* the transformation $L_g : h \mapsto gh$, and *right multiplication* the transformation $R_g : h \mapsto hg$. Then, thanks to Lemma 39, the group convolution can be rewritten with the functional formulation:

$$s_1 *_\Gamma s_2 = \sum_{g \in \Gamma} s_1[g]\, L_g(s_2) \tag{31}$$

$$= \sum_{g \in \Gamma} s_2[g]\, R_g(s_1) \tag{32}$$

which would then be commutative if, and only if, Γ were abelian.
The group convolution perserves the characterization from Theorem 41.

Theorem 44. Characterization of group convolution operators
Let a group Γ, let $f \in \mathcal{L}(\mathcal{S}(\Gamma))$,

(i) f is a group convolution right operator $\Leftrightarrow$ f is equivariant to left multiplications

(ii) f is a group convolution left operator $\Leftrightarrow$ f is equivariant to right multiplications

(iii) f is a group convolution commutative operator $\Leftrightarrow$ f is equivariant to multiplications

Proof. The proof is similar than the proof of Theorem 41.

$\Rightarrow$: Given $w \in \mathcal{S}(\Gamma)$, let $f_R = . *_\Gamma w$, $f_L = . *_\Gamma w$, $s \in \mathcal{S}(\Gamma)$, and $p \in \Gamma$. We have:

$$
\begin{aligned}
(L_p \circ f_R)(s) &= L_p(s *_\Gamma w) & (R_p \circ f_L)(s) &= R_p(w *_\Gamma s) \\
&= L_p\left(\sum_{g\in\Gamma} s[g]\, L_g(w)\right) & &= R_p\left(\sum_{g\in\Gamma} s[g]\, R_g(w)\right) \\
&= \sum_{g\in\Gamma} s[g]\, (L_p \circ L_g)(w) & &= \sum_{g\in\Gamma} s[g]\, (R_p \circ R_g)(w) \\
&= \sum_{pg\in\Gamma} s[p^{-1}pg]\, L_{pg}(w) & &= \sum_{gp\in\Gamma} s[gpp^{-1}]\, R_{gp}(w) \\
&= \sum_{g\in\Gamma} L_p(s)[g]\, L_g(w) & &= \sum_{g\in\Gamma} R_p(s)[g]\, R_g(w) \\
&= L_p(s) *_\Gamma w & &= w *_\Gamma R_p(s) \\
&= (f_R \circ L_p)(s) & &= (f_L \circ R_p)(s)
\end{aligned}
$$

$\Leftarrow$: We show the converse only for (i) since it is similar for (ii). Let $f \in \mathcal{L}(\mathcal{S}(\Gamma))$. We suppose $\forall p \in \Gamma, L_p \circ f = f \circ L_p$. First, note that

$$
\begin{aligned}
\forall h \in \Gamma, L_g(\delta_g)[h] = 1 &\Leftrightarrow \delta_g[g^{-1}h] = 1 \\
&\Leftrightarrow g^{-1}h = g \\
&\Leftrightarrow h = e \\
\textit{i.e. } L_g(\delta_g) &= \delta_e
\end{aligned}
$$

Then, we define $w = f(\delta_e)$. Let $s \in \mathcal{S}(\Gamma)$, we have:

$$\begin{aligned} f(s) &= \sum_{g \in \Gamma} s[g]\, f(\delta_g) \\ &= \sum_{g \in \Gamma} s[g]\, (f \circ L_g)(\delta_e) \\ &= \sum_{g \in \Gamma} s[g]\, (L_g \circ f)(\delta_e) \\ &= \sum_{g \in \Gamma} s[g]\, L_g(w) \\ &= s *_\Gamma w \end{aligned}$$

(iii) derives from (i) and (ii). □

2.2.2 Steered construction from groups

For a graph $G = \langle V, E \rangle$ and a subgroup $\Gamma \subset \Phi^*(V)$ of its invertible transformations, Definition 43 is applicable for $\mathcal{S}(\Gamma)$, but not for $\mathcal{S}(V)$ as V is not a group. Nonetheless, we will try to use the group convolution on $\mathcal{S}(\Gamma)$ to construct the convolutions on $\mathcal{S}(V)$. Our goal is to contruct a formulation like (31) while preserving Theorem 44 at the same time.

For now, let us assume that there is a subgroup $\Gamma \subset \Phi^*(V)$, which we associate through a one-to-one correspondence φ with V. We denote $\Gamma \overset{\varphi}{\cong} V$ and $g_v \overset{\varphi}{\mapsto} v$. Then, the linear morphism $\widetilde{\varphi}$ from $\mathcal{S}(\Gamma)$ to $\mathcal{S}(V)$ defined on the Dirac bases by $\widetilde{\varphi}(\delta_{g_v}) = \delta_{\varphi(g_v)}$ is a linear isomorphism. Hence $\mathcal{S}(V)$ inherits the same structural properties as $\mathcal{S}(\Gamma)$. For the sake of notational simplicity, we will use the same symbol φ for both φ and $\widetilde{\varphi}$ (as done between f and L_f). A commutative diagram between the sets is depicted on Figure 8.

$$\begin{array}{ccc} \Gamma & \xrightarrow{\varphi} & V \\ {\scriptstyle s}\downarrow & & \downarrow{\scriptstyle s} \\ \mathcal{S}(\Gamma) & \xrightarrow[\widetilde{\varphi}]{} & \mathcal{S}(V) \end{array}$$

Figure 8: Commutative diagram between sets

We naturally obtain the following relation, which put in simpler words means that signals on $\mathcal{S}(\Gamma)$ are mapped to $\mathcal{S}(V)$ when φ is simultaneously applied on both the signal space and its domain. At the same time, this relation is reminiscent of Lemma 39.

Lemma 45. Relation between $\mathcal{S}(\Gamma)$ and $\mathcal{S}(V)$

$$\forall s \in \mathcal{S}(\Gamma), \forall u \in V, \varphi(s)[u] = s[\varphi^{-1}(u)] = s[g_u]$$

Proof.

$$\begin{aligned} \forall s \in \mathcal{S}(\Gamma), \varphi(s) = \varphi(\sum_{g\in\Gamma} s[g]\,\delta_g) = \sum_{g\in\Gamma} s[g]\,\varphi(\delta_g) = \sum_{g\in\Gamma} s[g]\,\delta_{\varphi(g)} \\ = \sum_{v\in V} s[g_v]\,\delta_v \\ \text{So } \forall u \in V, \varphi(s)[u] = s[g_u] \end{aligned}$$

□

With this lemma, we can steer the definition of the group convolution from $\mathcal{S}(\Gamma)$ to $\mathcal{S}(V)$ as in the following definition.

Remark. Note that the following definition is just a step in our construction since we will see that it is not enough to obtain the wanted counterpart of Theorem 41.

Definition 46. Vertex-group convolution
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$. The vertex-group convolution between two signals s_1 and $s_2 \in \mathcal{S}(V)$ is defined as:

$$\forall u \in V, (s_1 *_\Gamma s_2)[u] = \sum_{v \in V} s_1[v]\, s_2[\varphi(g_v^{-1} g_u)]$$

We use the same symbol $*_\Gamma$ since the group and vertex-group convolution are the same operation but applied to different domains.

Lemma 47. Relation between group and vertex-group convolutions
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$,

$$\forall s_1, s_2 \in \mathcal{S}(\Gamma), \forall u \in V, (\varphi(s_1) *_\Gamma \varphi(s_2))[u] = (s_1 *_\Gamma s_2)[g_u]$$

Proof. Using Lemma 45,

$$\begin{aligned}
(\varphi(s_1) *_\Gamma \varphi(s_2))[u] &= \sum_{v \in V} \varphi(s_1)[v]\, \varphi(s_2)[\varphi(g_v^{-1} g_u)] \\
&= \sum_{v \in V} s_1[g_v]\, s_2[g_v^{-1} g_u] \\
&= \sum_{g \in \Gamma} s_1[g]\, s_2[g^{-1} g_u] \\
&= (s_1 *_\Gamma s_2)[g_u]
\end{aligned}$$

□

Let f be a convolution right operator on $\mathcal{S}(V)$, *s.t.* $f = . *_\Gamma w$, and $\widetilde{f}$ be its

counterpart on $\mathcal{S}(V)$, *i.e.* $\widetilde{f} = . *_\Gamma \varphi^{-1}(w)$. Then, we have

$$\begin{aligned} \forall s \in \mathcal{S}(V), f(s)[u] &= \widetilde{f}(\varphi^{-1}(s))[g_u] && \text{(Lemma 47)} \\ &= \varphi(\widetilde{f}(\varphi^{-1}(s)))[u] && \text{(Lemma 45)} \\ \textit{i.e. } f &= \varphi \circ \widetilde{f} \circ \varphi^{-1} && (33) \end{aligned}$$

This is also better depicted on a commutative diagram, see Figure 9.

$$\begin{array}{ccc} \mathcal{S}(\Gamma) & \xleftarrow{\varphi^{-1}} & \mathcal{S}(V) \\ {\scriptstyle \widetilde{f}}\downarrow & & \downarrow{\scriptstyle f} \\ \mathcal{S}(\Gamma) & \xrightarrow[\varphi]{} & \mathcal{S}(V) \end{array}$$

Figure 9: Commutative diagram of (33)

And so, given $p \in \Gamma$, the equivariance of $\widetilde{f}$ rewrites for f:

$$\begin{aligned} \varphi \circ L_p \circ \widetilde{f} \circ \varphi^{-1} &= \varphi \circ \widetilde{f} \circ L_p \circ \varphi^{-1} && (34) \\ \textit{i.e. } (\varphi \circ L_p \circ \varphi^{-1}) \circ f &= f \circ (\varphi \circ L_p \circ \varphi^{-1}) && (35) \end{aligned}$$

That is, if we paraphrase Theorem 41, the class of vertex-group convolution operators is the class of linear operators that are equivariant to $\{\varphi \circ L_p \circ \varphi^{-1}, p \in \Gamma\}$. However, our goal is to obtain equivariance to Γ. Since L_p is not exactly an object of Γ, but a realization of a group action, we are going to consider group actions next.

2.2.3 Construction under group actions

We have been previously using the notion of group action implicitly, let us define it here.

Definition 48. Group action
An *action* of a group Γ on a set V is a homomorphism $L : \Gamma \to \Phi^*(V)$.

Remark. Given $g \in \Gamma$, we denote $L_g = L(g)$, or just $g(.)$ when there is no ambiguity. L_g is called the action of g by L on V. When a group Γ is in one-to-one correspondence with a set V, we will abusively attribute *actions* of the objects of the group to their corresponding objects in the set.

Hence, note that a transformation $g \in \Gamma$ can act on both Γ through the left multiplication L and on V as itself. This ambivalence can be seen on a commutative diagram, see Figure 10.

$$\begin{array}{ccc} g_u & \xrightarrow{L_{g_v}} & g_v g_u \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle\varphi} \\ u & \xrightarrow[g_v]{(36)} & \varphi(g_v g_u) \end{array}$$

Figure 10: Commutative diagram. All arrows except for the one labeled with (36) are always true.

For (36) to be true means that φ is an equivariant map *i.e.* whether the mapping is done before or after the action of Γ has no impact on the result. When such φ exists, Γ and V are said to be equivalent and we denote $\Gamma \equiv V$.

Definition 49. Equivariant map
A map φ from a group Γ acting on the destination set V and itself, is said to be an *equivariant map* if

$$\forall g, h \in \Gamma, g(\varphi(h)) = \varphi(g(h))$$

Remark. For example, if g acts on Γ through left multiplication then $g(h) = L_g(h) = gh$.

Suppose we have $\Gamma \overset{\varphi}{\cong} V$. If we also have that $\Gamma \equiv V$, we are interested to know if then φ exhibits the equivalence and under which auto-action.

Definition 50. φ-Equivalence
A subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$, is said to be *(left) φ-equivalent* if φ is a bijective equivariant map under left multiplication *i.e.* :

$$\forall v, u \in V, g_v(u) = \varphi(g_v g_u) \tag{36}$$
$$\textit{i.e. } \forall g \in \Gamma, g = \varphi \circ L_g \circ \varphi^{-1} \tag{37}$$

It is said to be *right φ-equivalent* if φ is a bijective equivariant map under right multiplication *i.e.* :

$$\forall v, u \in V, g_v(u) = \varphi(g_u g_v) \tag{38}$$
$$\textit{i.e. } \forall g \in \Gamma, g = \varphi \circ R_g \circ \varphi^{-1} \tag{39}$$

We denote $\Gamma \overset{\varphi}{\equiv} V$. Unless stated otherwise, we will implicitly consider it is under left multiplication.

Remark. For example, translations on the grid graph, with $\varphi(t_{i,j}) = (i, j)$, are φ-equivalent since $t_{i,j}(a, b) = \varphi(t_{i,j} \circ t_{a,b})$. However, with $\varphi(t_{i,j}) = (\text{-}i, \text{-}j)$, they would not be φ-equivalent since in general $t_{i,j}(a, b) \neq t_{i,j}(\text{-}a, \text{-}b)$.

We are now equipped to define the vertex-action convolution.

Definition 51. Vertex-action convolution
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$. The vertex-action convolution between two signals s_1 and $s_2 \in \mathcal{S}(V)$ is defined as:

$$s_1 *_V s_2 = \sum_{v \in V} s_1[v]\, g_v(s_2) \tag{40}$$

$$= \sum_{g \in \Gamma} s_1[\varphi(g)]\, g(s_2) \tag{41}$$

The two expressions differ on the domain upon which the summation is done. Expression (40) puts the emphasis on each vertex and its action, whereas expression (41) puts the emphasis on each transformation of Γ.

Lemma 52. Relation with vertex-group convolution
There is φ-equivalence if, and only if, vertex-group and vertex-action convolutions are equal, *i.e.* $\Gamma \overset{\varphi}{\equiv} V \Leftrightarrow *_\Gamma = *_V$

Proof.

$$\begin{aligned} &\forall s_1, s_2 \in \mathcal{S}(V), \\ &\quad s_1 *_\Gamma s_2 = s_1 *_V s_2 \\ &\quad \Leftrightarrow \forall u \in V, \sum_{v \in V} s_1[v]\, s_2[\varphi(g_v^{-1} g_u)] = \sum_{v \in V} s_1[v]\, s_2[g_v^{-1}(u)] \end{aligned} \tag{42}$$

Hence, the direct sense is obtained by applying (36).
For the converse, given $u, v \in V$, we first realize (42) for $s_1 := \delta_v$, obtaining $s_2[\varphi(g_v^{-1} g_u)] = s_2[g_v^{-1}(u)]$, which we then realize for a real signal s_2 having

no two equal entries, obtaining $\varphi(g_v^{-1}g_u) = g_v^{-1}(u)$. From the latter we finally obtain (36) with the one-to-one correspondence $g_{v'} := g_v^{-1}$. □

Remark. Note that we used the fact that the φ-equivalence is implicitly left. If it were right, we would have instead $s_1 *_\Gamma s_2 = s_2 *_V s_1$.

We now coin the term φ-convolution, for when there is φ-equivalence.

Definition 53. φ-convolution
The *φ-convolution* is defined as the vertex-action convolution such that $\Gamma \overset{\varphi}{\equiv} V$, *i.e.* given s_1 and $s_2 \in \mathcal{S}(V)$, we have:

$$\begin{aligned} s_1 *_\varphi s_2 &= s_1 *_V s_2 \\ &= s_1 *_\Gamma s_2 \end{aligned}$$

This time we do obtain equivariance to Γ as expected, and the full characterization as well.

Theorem 54. Characterization of φ-convolution right operators
Let a subgroup $\Gamma \subset \Phi^*(V)$ *s.t.* $\Gamma \overset{\varphi}{\equiv} V$, let $f \in \mathcal{L}(\mathcal{S}(\Gamma))$, then:

f is a φ-convolution right operator $\Leftrightarrow$ f is equivariant to Γ

Proof. As a consequence of (35), (37), Lemmas 47, and 52, we can use bullet (i) of Theorem 44 to obtain the proof. □

Corollary 55. Characterization of φ-convolution operators
Let a subgroup $\Gamma \subset \Phi^*(V)$ *s.t.* $\Gamma \overset{\varphi}{\equiv} V$, let $f \in \mathcal{L}(\mathcal{S}(\Gamma))$, then:

f is equivariant to $\Gamma \Leftrightarrow$ f is a φ-convolution operator *s.t.* its laterality is opposed to the laterality of the φ-equivalence

Proof. As per Theorem 54 and its counterpart for bullet (ii) of Theorem 44. □

Remark. The bullet (iii) would just inform that it is equivalent to say that either the φ-equivalence is commutative, or the φ-convolution is commutative, or Γ is abelian.

Theorem 54 tells us that in order to define a convolution on the vertex domain of a graph $G = \langle V, E \rangle$, all we need is a subgroup Γ of invertible transformations of the vertex set V (or of a subset of V), that is equivalent to it. The choice of Γ can be done with respect to the edge set E. This is discussed in more details in Section 2.3, where we will see that in fact, we only need a generating set of Γ.
The previous construction relies on exposing a bijective equivariant map φ between Γ and V. In the next subsection, we show that in case Γ is abelian, we even need not expose φ and the characterization still holds.

2.2.4 Mixed domain formulation

From (41), we can define a mixed domain convolution *i.e.* that is defined for $r \in \mathcal{S}(\Gamma)$ and $s \in \mathcal{S}(V)$, without the need of expliciting φ.

Definition 56. Mixed domain convolution
Let a subgroup $\Gamma \subset \Phi^*(V)$. The *mixed domain convolution* between two signals $r \in \mathcal{S}(\Gamma)$ and $s \in \mathcal{S}(V)$ results in a signal $r *_{\text{M}} s \in \mathcal{S}(V)$ and is defined as:

$$r *_m s = \sum_{g \in \Gamma} r[g]\, g(s)$$

We call it m-convolution. From a practical point of view, this expression of the convolution is useful because it relegates φ as an underpinning object.

Lemma 57. Relation with vertex-action convolution
$\forall \varphi \in \textsc{bij}(\Gamma, V), \forall (r, s) \in \mathcal{S}(\Gamma) \times \mathcal{S}(V)$,

$$r *_m s = \varphi(r) *_V s$$

Proof. Let $\varphi \in \textsc{bij}(\Gamma, V), (r, s) \in \mathcal{S}(\Gamma) \times \mathcal{S}(V)$,

$$\begin{aligned} r *_m s &= \sum_{g \in \Gamma} r[g]\, g(s) = \sum_{v \in V} r[g_v]\, g_v(s) \overset{(\diamond)}{=} \sum_{v \in V} \varphi(r)[v]\, g_v(s) \\ &= \varphi(r) *_V s \end{aligned}$$

Where $\overset{(\diamond)}{=}$ comes from Lemma 45. □

In other words, $*_m$ is a convenient reformulation of $*_V$.

Lemma 58. Relation with group, vertex-group and φ-convolutions
Let $\varphi \in \textsc{bij}(\Gamma, V), (r, s) \in \mathcal{S}(\Gamma) \times \mathcal{S}(V)$, we have:

$$\begin{aligned} \Gamma \overset{\varphi}{\equiv} V &\Leftrightarrow \forall v \in V, (r *_m s)[v] = (r *_\Gamma \varphi^{-1}(s))[g_v] \\ &\Leftrightarrow r *_m s = \varphi(r) *_\Gamma s \\ &\Leftrightarrow r *_m s = \varphi(r) *_\varphi s \end{aligned}$$

Proof. On one hand, Lemma 57 gives $r *_m s = \varphi(r) *_V s$. On the other hand, Lemma 47 gives $\forall v \in V, (r *_\Gamma \varphi^{-1}(s))[g_v] = (\varphi(r) *_\Gamma s)[v]$. Then Lemma 52 concludes. □

From m-convolution, a left operator can be used as a layer of a neural network since it is defined over $\mathcal{S}(V)$, whereas right operators are not.

Proposition 59. Condition of equivariance
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \cong V$. Γ is abelian, if and only if, m-convolution left operators are equivariant to Γ.

Proof. Let $w, g \in \Gamma$, and define $f_w : s \mapsto w *_m s$. In the following expressions, Γ is abelian if and only if (43) and (44) are equal (the converse is obtained by particularizing on well chosen signals):

$$
\begin{aligned}
(f_w \circ g)(s) &= \sum_{h \in \Gamma} w[h]\, hg(s) && (43)\\
&= \sum_{h \in \Gamma} w[h]\, gh(s) && (44)\\
&= g\left(\sum_{h \in \Gamma} w[h]\, h(s) \right)\\
&= g(w *_m s)\\
&= (g \circ f_w)(s)
\end{aligned}
$$

□

Remark. We used this proof because it is simpler. However, the reason behind the abelian condition is that mixed domain convolution are expressed as if the underlying φ-equivalence were left. Therefore it is a consequence of Corrolary 55. Note that if we used a right expression for $*_m$ (which amounts to commute the operands), we would be interested in right operators instead of left ones, which would just change the problem to "right-right" instead of "left-left" which would also require abelianity for the equivariance.

We then coin the term M-convolution, for which Γ is abelian.

Definition 60. M-convolution
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \cong V$.
The M-*convolution* is defined as a mixed domain convolution such that Γ is an abelian subgroup of $\Phi^*(V)$. We denote it $*_{\text{M}}$.

Corollary 61. Characterization of M-convolution left operators
Let a subgroup $\Gamma \subset \Phi^*(V)$ *s.t.* $\Gamma \cong V$, and Γ is abelian. Let $f \in \mathcal{L}(\mathcal{S}(\Gamma))$, then:

$$f \text{ is a M-convolution left operator} \Leftrightarrow f \text{ is equivariant to } \Gamma$$

Proof. As a corollary of Corrolary 55 and Proposition 59. □

Remark. Note that despite abelianity, we do not call them "commutative" operators but still call "left" operators since the domain is mixed.

When the subgroup domain is abelian, it is preferable to use $*_{\text{M}}$ rather than $*_\varphi$ since φ is not needed.
The obtained contructions φ- and M-convolutions are independent of the edge set. In fact, they characterize more generally convolutions for signals defined on sets. Next, we will see what happens should we also consider the edge set.

2.3 Inclusion of the edge set in the construction

The constructions from the previous section involve the vertex set V and depend on Γ, a subgroup of the vertex symmetric group. Therefore, it looks natural to try to relate the edge set and Γ.
There are two point of views. Either Γ describes an underlying graph structure $G = \langle V, E \rangle$, either G can be used to define a relevant subgroup Γ to which the produced convolutive operators will be equivariant. Both approaches will help characterize classes of graphs that can support natural definitions of convolutions.

2.3.1 Edge-constrained convolutions

In this subsection, we are trying to answser the following question:

- What graphs admit a φ-convolution, or an M-convolution (in the sense that they can be defined with the characterization), under the condition that Γ is generated by a set of edge-constrained transformations ?

Definition 62. Edge-constrained transformation
An *edge-constrained* (EC) transformation on a graph $G = \langle V, E \rangle$ is a transformation $f : V \mapsto V$ such that

$$\forall u, v \in V, f(u) = v \Rightarrow u \overset{E}{\sim} v$$

We denote $\Phi_{\text{EC}}(G)$ and $\Phi^*_{\text{EC}}(G)$ the sets of EC and invertible EC transformations. When a convolution is defined as a sum over a set that is in one-to-one correspondence with a group that is generated from a set of EC transformations, we call it an EC convolution.

Remark. Note that $\Phi^*_{\text{EC}}(G)$ is not a group, that is why we are interested in groups and their generating sets.

This leads us to consider Cayley graphs (Cayley, 1878).

Definition 63. Cayley graph
Let a group Γ and one of its generating set $\mathcal{U}$. The *Cayley graph* generated by $\mathcal{U}$, is the digraph $\vec{G} = \langle V, E \rangle$ such that $V = \Gamma$ and E is such that, either:

- $\forall a, b \in \Gamma, a \to b \Leftrightarrow \exists g \in \mathcal{U}, ga = b$ (*left Cayley graph*)
- $\forall a, b \in \Gamma, a \to b \Leftrightarrow \exists g \in \mathcal{U}, ag = b$ (*right Cayley graph*)

In case Γ is abelian, we call it an *abelian Cayley graph*. Also, we call *Cayley subgraph* a subgraph that is isomorph to a Cayley graph.

Remark. Note that in the literature, a Cayley graph is usually what we defined as a right one. In what follows, we could have used right Cayley graphs instead of left ones, but since we want g to be an EC transformation, we would have had to use the group action $g(a) = L_g(a) = ag$ for the φ-convolution, which is inconvenient with the functional notation. Recall that instead, we defined φ-equivalence with a group action corresponding to left multiplication.

Convolution on Cayley graphs
In the case of left Cayley graphs, it is clear that $\mathcal{U} \subseteq \Phi^*_{\text{EC}}$ and $\Phi^* \supseteq \langle \mathcal{U} \rangle \equiv V$. So that they admit EC φ-convolutions, and EC M-convolutions in the abelian case.
More precisely, we obtain the following characterization:

Theorem 64. EC characterization by left Cayley subgraph
Let a graph $G = \langle V, E \rangle$, then:

(i) G admits an EC φ-convolution if and only if it contains a left Cayley subgraph

(ii) G admits an EC M-convolution if and only if it contains an abelian Cayley subgraph

Proof. TODO: Vincent peux-tu vérifier cette preuve stp?

We show the result only in the general case as the proof for the abelian case is similar.

1. From left to right: as a direct application of the definitions.

2. From right to left:
   Let a graph $G = \langle V, E \rangle$. We suppose it contains a subgraph $\vec{G_s} = \langle V_s, E_s \rangle$ that is graph-isomorph to a left Cayley graph $\vec{G_c} = \langle V_c, E_c \rangle$, generated by $\mathcal{U}$. Let ψ be a graph isomorphism from G_s to G_c. To obtain the proof, we need to find a group of invertible transformations Γ of V_s generated by a set of EC transformations, such that $\Gamma \equiv V_s$.

   Let's define the group action $L : V_c \times V_s \to V_s$ inductively as follows:

   (a) $\forall g \in \mathcal{U}, L_g(u) = v \Leftrightarrow g\psi(u) = \psi(v)$

   (b) Whenever L_g and L_h are defined, the action of gh is defined by homomorphism as $L_{gh} = L_g \circ L_h$

   (c) Whenever L_g is defined, the action of g^{-1} is defined by homomorphism as $L_{g^{-1}} = L_g^{-1}$ *i.e.* $L_{g^{-1}}(u) = v \Leftrightarrow \psi(u) = g\psi(v)$

   Note that the induction transfers the property (a) to all $g \in V_c$ in a transitive manner because

$$L_{gh}(u) = L_g(L_h(u)) = w \Leftrightarrow \exists v \in V_s \begin{cases} L_h(u) = v \\ L_g(v) = w \end{cases}$$

and

$$\exists v \in V_s \begin{cases} h\psi(u) = \psi(v) \\ g\psi(v) = \psi(w) \end{cases} \Leftrightarrow gh\psi(u) = \psi(w)$$

We must also verify that this construction is well-defined, *i.e.* whenever we define an action with (b) or (c), if the action was already defined, then they must be equal. This is the case because the homomorphism $g \mapsto L_g$ on V_c is in fact an isomorphism as

$$\begin{aligned} L_g = L_h &\Leftrightarrow \forall u \in V, L_g(u) = L_h(u) \\ &\Leftrightarrow \forall u \in V, g\psi(u) = h\psi(u) \\ &\Leftrightarrow g = h \end{aligned}$$

Also note that (c) is needed only in case that V_c is infinite.

Denote the set $L_{\mathcal{U}} = \{L_g, g \in \mathcal{U}\}$ and $\Gamma = \langle L_{\mathcal{U}} \rangle \cong V_c$. Let's define the map φ as:

$$\begin{aligned} \Gamma &\to V_s \\ \varphi : L_g &\mapsto L_g(\psi^{-1}(\mathrm{Id})) \end{aligned}$$

φ is bijective because $\forall g \in V_c, \varphi(L_g) = \psi^{-1}(g)$ thanks to (a).

Additionally, we have:

$$\begin{aligned} L_h(\varphi(L_g) &= L_h(L_g(\psi^{-1}(\mathrm{Id}))) \\ &= L_h \circ L_g(\psi^{-1}(\mathrm{Id})) \\ &= L_{hg}(\psi^{-1}(\mathrm{Id})) \\ &= \varphi(L_{hg}) \\ &= \varphi(L_h \circ L_g) \end{aligned}$$

That is, φ is a bijective equivariant map and $\langle L_{\mathcal{U}} \rangle = \Gamma \overset{\varphi}{\equiv} V_s$. Moreover, $L_{\mathcal{U}}$ is a set of EC transformations thanks to (a). Therefore, G admits an EC φ-convolution.

□

Corollary 65. Characterization by φ
Let a graph $G = \langle V, E \rangle$, and a set $\mathcal{U} \subset \Phi^*_{\text{EC}}(G)$ *s.t.*

$$\langle \mathcal{U} \rangle \cong \Gamma \equiv V' \subset V$$

G admits an EC φ-convolution, if and only if, φ is a graph isomorphism between the left Cayley graph generated by $\mathcal{U}$ and the subgraph induced by V'.

The proof is omitted as it would be highly similar to the previous one.

2.3.2 Intrinsic properties

- Obviously the constructed convolutions are linear. But do they also preserve the locality and weight sharing properties ?

Recall that a φ-convolution operator is a right operator, and can be expressed as

$$\forall s \in \mathcal{S}(V), \forall u \in V,$$
$$f_w(s)[u] = (s *_\varphi w)[u] = \sum_{v \in V} s[v]\, w[g_v^{-1}(u)]$$

From this expression, it is not obvious that f_w is a local operator. First we must define a notion of locality.

Definition 66. Compact (of a graph)
Let a graph $G = \langle V, E \rangle$. A subset of the vertex set $\mathcal{K} \in V$ is said to be *compact* if its induced subgraph in G is connected.

Roughly speaking, we want for our notion of locality that when the support of w is small and compact, then $f_w(s)[u]$ should be defined by a sum of a few corresponding v too. This translates into the following definition.

Definition 67. Compactly supported φ-convolution operator
A φ-convolution operator f_w is said to be Compactly Supported (CS) if

1. $\mathrm{supp}(w)$ is a compact, and
2. given $u \in V$, there is a graph isomorphism between $\mathrm{supp}(w)$ and $\mathrm{supp}(\sigma_u)$, where $\sigma_u : v \to s[v]\, w[g_v^{-1}(u)]$ (so that $f_w(s)[u] = \sum_{v \in V} \sigma_u(v)$)

Before characterizing CS convolution operators, we demonstrate this lemma.

Lemma 68. On right Cayley graphs, left multiplication operators are automorphisms (and vice-versa).

Proof. Since $a \to b \Leftrightarrow \exists g, ag = b \Leftrightarrow \exists g, hag = hb \Leftrightarrow ha \to hb$. □

We obtain the following theorem, which is the lateral symmetry of the formulation of Theorem 64.

Theorem 69. CS characterization by right Cayley subgraph
Let a graph $G = \langle V, E \rangle$, then:

(i) G admits a CS φ-convolution if and only if it contains a right Cayley subgraph

(ii) G admits a CS M-convolution if and only if it contains an abelian Cayley subgraph

Proof. We only demonstrate the first case as the proof in the abelian case is similar.
TODO: Vincent, peux-tu m'aider stp à démontrer cette preuve à l'aide du lemme 66, quitte à modifier la définition 65?

□

Let's denote $\mathcal{K}_u = g_u(\varphi(\mathcal{N}^{-1})) \subset V$. Let's define $\mathcal{M}$, $\mathcal{N}$ as in the previous proof. We call $\mathcal{N}$ the *supporting set* of the convolution.

Definition 70. Supporting set
The *supporting set* of an EC convolution operator f_w, is a set $\mathcal{N} \subset \Phi^*_{\text{EC}}$, such that

(i) when $*$ is $*_\varphi$: $0 \notin w[\mathcal{M}]$, where $\mathcal{M} = \varphi(\mathcal{N})$

(ii) when $*$ is $*_{\text{M}}$: $0 \notin w[\mathcal{N}]$

Definition 71. Local patch for $*_\varphi$
The *local patch* at $u \in V$ of an EC φ-convolution operator f_w is defined as $\mathcal{K}_u = g_u(\varphi(\mathcal{N}^{-1}))$.

Remark. In other terms, $\mathcal{K}_{\text{Id}} = \varphi(\mathcal{N}^{-1})$ is the *initial local patch*, which is composed of all vertices that are connected in direction to $\varphi(\text{Id})$; and $\mathcal{K}_u$ is obtained by moving $\mathcal{K}_{\text{Id}}$ on the Cayley subgraph via the edges corresponding to the decomposition of g_u on the generating set $\mathcal{U}$.

To see that the weights are tied in the general case (i), we can show the following proposition.

Proposition 72. Weight sharing
$\forall a, \alpha \in V, \forall b \in \mathcal{K}_a : \exists \beta \in \mathcal{K}_\alpha \Leftrightarrow g_\beta^{-1}(\alpha) = g_b^{-1}(a)$

Proof. By using (36),

$$\begin{aligned} g_{\mathcal{K}_\alpha}^{-1}(\alpha) = g_{\mathcal{K}_a}^{-1}(a) &\Leftrightarrow g_\alpha^{-1} g_{\mathcal{K}_\alpha} = g_a^{-1} g_{\mathcal{K}_a} \\ &\Leftrightarrow \mathcal{K}_\alpha = g_\alpha g_a^{-1}(\mathcal{K}_a) = g_\alpha g_a^{-1} g_a(\varphi(\mathcal{N}^{-1})) \\ &\Leftrightarrow \mathcal{K}_\alpha = g_\alpha(\varphi(\mathcal{N}^{-1})) \end{aligned}$$

□

2.3.3 Stricly edge-constrained convolutions

We make the disctinction between general EC convolution operators and those for which the weight kernel w is smaller and is supported only on EC transformations of $\mathcal{U}$.

Definition 73. Strictly EC convolution operator
A *strictly* edge-constrained (EC*) convolution operator f_w, is an EC convolution operator such that its supporting set $\mathcal{N} \subset \mathcal{U}$.

Remark. EC* convolution operators are simpler to obtain as we can construct them just with $\mathcal{U} \subset \Phi_{\text{EC}}^*(G)$ without composing the transformations.

Let f_w be an EC* convolutional operator. In the general case (i), $w \in \mathcal{S}(V)$, so its support is $\mathcal{M} = \varphi(\mathcal{N})$ such that $\mathcal{N} \subseteq \mathcal{U}$. In the abelian case (ii), we use instead $w \in \mathcal{S}(\Gamma)$, and thus its support is directly $\mathcal{N}$. Therefore, we can rewrite the expressions of the convolution operator as:

(i) $\forall s \in \mathcal{S}(V), \forall u \in V, f_w(s)[u] \overset{(\varphi)}{=} \sum_{v \in \mathcal{K}_u} s[v]\, w[g_v^{-1}(u)]$

(ii) $\forall s \in \mathcal{S}(V), f_w(s) \overset{(\text{M})}{=} \sum_{g \in \mathcal{N}} w[g]\, g(s)$

Remark. Note that in the abelian case, we can see from (ii) that a definition of a local patch would coincide with the supporting set, so that locality and weight sharing is straightforward.

Construction

From these expressions, it is clear that Γ needs not to be fully determined to calculate $f_w(s)[u]$. The case (ii) is the simplest as the only requirement is a supporting set $\mathcal{N}$ of EC invertible transformations. In the case (i), we also need to determine $\mathcal{K}_u$.

Strict locality

Note that $f_w(s)[u]$ is a weighted aggregation of entries $s[v]$ for $v \in \mathcal{K}_u$. As $\mathcal{K}_{\mathrm{Id}} = \varphi(\mathcal{N}^{-1}) = \mathcal{N}^{-1}(\varphi(\mathrm{Id}))$, $\mathcal{K}_{\mathrm{Id}}$ contains only neighbors of $\varphi(\mathrm{Id})$, and so $\mathcal{K}_u = g_u(\mathcal{K}_{\mathrm{Id}})$ contains only neighbors of u. Therefore, in both cases $f_w(s)[u]$ is a weighted aggregation of entries located in the neighborhood of u.

Complexity

Another merit is that EC* convolutions have a complexity of $\mathcal{O}(kn)$, where $n = |V|$ is the order of the graph, and $k = |\mathcal{N}|$ is the size of the weight kernel. In comparison, EC convolutions have complexity up to $\mathcal{O}(n^2)$.

2.4 From groups to groupoids

2.4.1 Motivation

One possible limitation coming from searching for Cayley subgraphs is that they are degree-regular *i.e.* the in- and the out-degree $d = |\mathcal{U}|$ of each vertex is the same. That is, for a general graph G, the size of the weight kernel w of an EC* convolution operator f_w supported on $\mathcal{U}$ is bounded by d, which in turn is bounded by twice the mininimal degree of G (twice because G is undirected and $\mathcal{U}$ can contain every inverse).
There are a lot of possible strategies to overcome this limitation. For example:

1. connecting each vertex with its k-hop neighbors, with $k > 1$,
2. artificially creating new connections for less connected vertices,
3. ignoring less connected vertices,
4. allowing the supporting set $\mathcal{N}$ to exceed $\mathcal{U}$ *i.e.* dropping * in EC*.

These strategies require to concede that the topological structure supported by G is not the best one to support an EC* convolution on it, which breeds the following question:

- What can we relax in the previous EC* contruction in order to unbound the supporting set, and still preserve the equivariance characterization?

The latter constraint is a consequence that every vertex of the Cayley subgraph $\vec{G}$ must be composable with every generator from $\mathcal{U}$. Therefore, an answer consists in considering groupoids (Brandt, 1927) instead of groups. Roughly speaking, a groupoid is almost a group except that its composition law needs not be defined everywhere. Weinstein, 1996, unveiled the

benefits to base convolutions on groupoids instead of groups in order to exploit partial symmetries.

2.4.2 Definition of notions related to groupoids

Definition 74. Groupoid
A *groupoid* Υ is a set equipped with a partial composition law with domain $\mathcal{D} \subset \Upsilon \times \Upsilon$, called *composition rule*, that is

1. closed into Υ *i.e.* $\forall (g,h) \in \mathcal{D}, gh \in \Upsilon$
2. associative *i.e.* $\forall f,g,h \in \Upsilon,$ $\begin{cases} (f,g),(g,h) \in \mathcal{D} \Leftrightarrow (fg,h),(f,gh) \in \mathcal{D} \\ (f,g),(fg,h) \in \mathcal{D} \Leftrightarrow (g,h),(f,gh) \in \mathcal{D} \\ \text{when defined, } (fg)h = f(gh) \end{cases}$
3. invertible *i.e.* $\forall g \in \Upsilon, \exists! g^{-1} \in \Upsilon \quad s.t. \begin{cases} (g,g^{-1}),(g^{-1},g) \in \mathcal{D} \\ (g,h) \in \mathcal{D} \Rightarrow g^{-1}gh = h \\ (h,g) \in \mathcal{D} \Rightarrow hgg^{-1} = h \end{cases}$

Optionally, it can be *domain-symmetric i.e.* $(g,h) \in \mathcal{D} \Leftrightarrow (h,g) \in \mathcal{D}$, and *abelian i.e.* domain-symmetric with $gh = hg$.

Remark. Note that left and right inverses are necessarily equal (because $(gg^{-1})g = g(g^{-1}g)$). Also note we can define a right identity element $e_g^r = g^{-1}g$, and a left one $e_g^l = gg^{-1}$, but they are not necessarily equal and depend on g.

Most definitions related to groups can be adapted to groupoids. In particular, let's adapt a few notions.

Definition 75. Groupoid partial action
A partial *action* of a groupoid Υ on a set V, is a function L, with domain $\mathcal{D}_L \subset \Upsilon \times V$ and valued in V, such that the map $g \mapsto L_g$ is a groupoid homomorphism.

Remark. As usual, we will confound L_g and g when there is no possible confusion, and we denote $\mathcal{D}_{L_g} = \mathcal{D}_g = \{v \in V, (g, v) \in \mathcal{D}_L\}$.

Definition 76. Partial equivariant map
A map φ from a groupoid Υ partially acting on the destination set V is said to be a *partial equivariant map* if

$$\forall g, h \in \Upsilon, \begin{cases} \varphi(h) \in \mathcal{D}_g \Leftrightarrow (g, h) \in \mathcal{D} \\ g(\varphi(h)) = \varphi(gh) \end{cases}$$

Also, φ-equivalence between a subgroupoid and a set is defined similarly with φ being a bijective *partial* equivariant map between them.

Definition 77. Partial transformations groupoid
The *partial transformations groupoid* $\Psi^*(V)$, is the set of invertible partial transformations, equipped with the functional composition law with domain $\mathcal{D}$ such that

$$\begin{cases} \mathcal{D}_{gh} = h(\mathcal{D}_h) \cap \mathcal{D}_g \\ (g, h) \in \mathcal{D} \Leftrightarrow \mathcal{D}_{gh} \neq \emptyset \end{cases}$$

Remark. Note that a subgroupoid $\Upsilon \subset \Psi^*(V)$ is domain-symmetric when $\exists v \in V, g(v) \in \mathcal{D}_h \Leftrightarrow \exists u \in V, h(u) \in \mathcal{D}_g$

2.4.3 Construction of partial convolutions

The expression of the convolution we constructed in the previous section cannot be applied as is. We first need to extend the algebraic objects we work with. Extending a partial transformation g on the signal space $\mathcal{S}(V)$ (and thus the convolutions) is a bit tricky, because only the signal entries corresponding to $\mathcal{D}_g$ are moved. A convenient way to do this is to consider the groupoid closure obtained with the addition of an absorbing element.

Definition 78. Zero-closure
The *zero-closure* of a groupoid Υ, denoted Υ^0, is the set $\Upsilon \cup 0$, such that the groupoid axioms 1, 2 and 3, and the domain $\mathcal{D}$ are left unchanged, and

4. the composition law is extended to $\Upsilon^0 \times \Upsilon^0$ with $\forall (g,h) \notin \mathcal{D}, gh = 0$

Remark. Note that this is coherent as the properties 2 and 3 are still partially defined on the original domain $\mathcal{D}$.

Now, we will also extend every other algebraic object used in the expression of the φ-convolution and the μ-convolution, so that we can directly apply our previous constructions.

Lemma 79. Extension of φ on V^0
Let a partial equivariant map $\varphi : \Upsilon \to V$. It can be extended to a (total) equivariant map $\varphi : \Upsilon^0 \to V^0 = V \cup \varphi(0)$, such that $\varphi(0) \notin V$, that we denote $0_V = \varphi(0)$, and such that

$$\forall g \in \Upsilon^0, \forall v \in V^0, g(v) = \begin{cases} \varphi(g g_v) & \text{if } g_v \in \mathcal{D}_g \\ 0_V & \text{else} \end{cases}$$

Proof. We have $\varphi(0) \notin V$ because φ is bijective. Additionaly, we must have $\forall (g,h) \notin \mathcal{D}, g(\varphi(h)) = \varphi(gh) = \varphi(0) = 0_V$. □

Remark. Note that for notational conveniency, we may use the same symbol 0 for 0_Υ, 0_V and $0_\mathbb{R}$.

Similarly to $\Phi^*(V)$, $\Psi^*(V)$ can also move signals of $\mathcal{S}(V)$.

Lemma 80. Extension of injective partial transformations to $\mathcal{S}(V)$
Let $g \in \Psi^*(V)$. Its extension is done in two steps:

1. g is extended to $V^0 = V \cup \{0_V\}$ as $g(v) = 0_V \Leftrightarrow v \notin \mathcal{D}_g$.
2. Under the convention $\forall s \in \mathcal{S}(V), s[0_V] = 0_\mathbb{R}$, g is extended via linear extension to $\mathcal{S}(V)$, and we have

$$\forall s \in \mathcal{S}(V), \forall v \in V, g(s)[v] = s[g^{-1}(v)]$$

Proof. Straightforward. □

With these extensions, we can obtain the partial φ- and M-convolutions related to Υ almost by substituting Υ^0 to Γ in Definition 53 and Definition 56.

Definition 81. Partial convolution
Let a subgroupoid $\Upsilon \subset \Psi^*(V)$, such that $\Upsilon \overset{\varphi}{\rightrightarrows} V$. The partial φ- and M-convolutions, based on Υ, are defined on its zero-closure, with the same expression as if Υ^0 were a subgroup, and by extension of φ and of the groupoid partial actions *i.e.*

(i) $\forall s, w \in \mathcal{S}(V), s *_\varphi w = \sum_{v \in V} s[v]\, g_v(w) = \sum_{g \in \Upsilon} s[\varphi(g)]\, g(w)$

(ii) $\forall (w, s) \in \mathcal{S}(\Upsilon) \times \mathcal{S}(V), w *_\mathrm{M} s = \sum_{g \in \Upsilon} w[g]\, g(s)$

Symmetrical expressions
Note that, as $\forall r, r[0] = 0$, the partial convolutions can also be expressed on the domain $\mathcal{D}$ with a convenient symmetrical expression:

(i) $\forall u \in V, (s *_{\varphi} w)[u] = \sum_{\substack{(g_a, g_b) \in \mathcal{D} \\ s.t.\ g_a g_b = g_u}} s[a]\, w[b]$

(ii) $\forall u \in V, (w *_{\mathrm{M}} s)[u] = \sum_{\substack{v \in \mathcal{D}_g \\ s.t.\ g(v) = u}} w[g]\, s[v]$

We obtain an equivariance characterization similar to Theorem 54 and Theorem 61.

Theorem 82. Characterization by equivariance to Υ
Let a subgroupoid $\Upsilon \subset \Psi^*(V)$, such that $\Upsilon \overset{\varphi}{\equiv} V$, with $*$ based on Υ.

1. Then,
   (i) partial φ-convolution right-operators are equivariant to Υ,
   (ii) if Υ is abelian, partial M-convolution left-operators are equiv to Υ.
2. Conversely,
   (i) if Υ is domain-symmetric, linear transformations of $\mathcal{S}(V)$ that are equivariant to Υ are partial φ-convolution right-operators,
   (ii) if Υ is abelian, they are also partial M-convolution left-operators.

Proof. (i) (a) Direct sense:
Using the symmetrical expressions, and the fact that $\forall r, r[0] =$

0, we have

$$\begin{aligned}
(f_w \circ g(s))[u] &= \sum_{\substack{(g_a,g_b)\in\mathcal{D} \\ s.t.\ g_a g_b = g_u}} g(s)[a]\, w[b] \\
&= \sum_{\substack{(g_a,g_b)\in\mathcal{D} \\ s.t.\ g_a g_b = g_u}} s[g^{-1}(a)]\, w[b] \\
&= \sum_{\substack{(g_a,g_b)\in\mathcal{D} \\ s.t.\ (g,g_a)\in\mathcal{D} \\ s.t.\ g g_a g_b = g_u}} s[a]\, w[b] \\
&= \sum_{\substack{(g_a,g_b)\in\mathcal{D} \\ s.t.\ (g,g_a)\in\mathcal{D} \\ s.t.\ g_a g_b = g^{-1} g_u = g_{\varphi(g^{-1} g_u)} = g_{g^{-1}(u)}}} s[a]\, w[b] \\
&= f_w(s)[g^{-1}(u)] \\
&= (g \circ f_w(s))[u]
\end{aligned}$$

(b) Converse:

Let $v \in V$. Denote $e^r_{g_v} = g_v^{-1} g_v$ the right identity element of g_v, and $e^r_v = \varphi(e^r_{g_v})$. We have that

$$g_v(e^r_v) = v$$
$$\text{So, } \delta_v = g_v(\delta_{e^r_v})$$

Let $f \in \mathcal{L}(\mathcal{S}(V))$ that is equivariant to Υ, and $s \in \mathcal{S}(V)$. Thanks to the previous remark we obtain that

$$\begin{aligned}
f(s) &= \sum_{v\in V} s[v]\, f(\delta_v) \\
&= \sum_{v\in V} s[v]\, f(g_v(\delta_{e^r_v})) \\
&= \sum_{v\in V} s[v]\, g_v(f(\delta_{e^r_v}))
\end{aligned}$$

$$= \sum_{v \in V} s[v]\, g_v(w_v) \tag{45}$$

where $w_v = f(\delta_{e_v^r})$. In order to finish the proof, we need to find w such that $\forall v \in V, g_v(w) = g_v(w_v)$.

Let's consider the equivalence relation $\mathcal{R}$ defined on $V \times V$ such that:

$$\begin{aligned} a\mathcal{R}b &\Leftrightarrow w_a = w_b \\ &\Leftrightarrow e_a^r = e_b^r \\ &\Leftrightarrow g_a^{-1} g_a = g_b^{-1} g_b \\ &\Leftrightarrow (g_b, g_a^{-1}) \in \mathcal{D} \\ &\Leftrightarrow (g_a^{-1}, g_b) \in \mathcal{D} \end{aligned} \tag{46}$$

with (46) owing to the fact that Υ is domain-symmetric.

Given $x \in V$, denote its equivalence class $\mathcal{R}(x)$. Under the hypothesis of the axiom of choice (Zermelo, 1904) (if V is infinite), define the set $\aleph$ that contains exactly one representative per equivalence class. Let $w = \sum_{n \in \aleph} w_n$. Then V is the disjoint union $V = \cup_{n \in \aleph} \mathcal{R}(n)$ and (45) rewrites:

$$\begin{aligned} \forall u \in V, f(s)[u] &= \sum_{n \in \aleph} \sum_{v \in \mathcal{R}(n)} s[v]\, g_v(w_n)[u] \\ &= \sum_{n \in \aleph} \sum_{v \in \mathcal{R}(n)} s[v]\, w_n[g_v^{-1}(u)] \\ &= \sum_{n \in \aleph} \sum_{v \in \mathcal{R}(n)} s[v]\, w[g_v^{-1}(u)] \\ &= (s *_\varphi w)[u] \end{aligned} \tag{47}$$

where (47) is obtained thanks to (46).

(ii) With symmetrical expressions, it is clear that the convolution is

abelian, if and only if, Υ is abelian. Then (i) concludes.

□

Inclusion of EC

Similarly to the construction in Section 2.3, partial convolutions can define EC and EC* counterparts with a characterization of admissibility by groupoid Cayley subgraph isomorphism, and similar intrinsic properties.

Limitation of partial convolutions

However, because of the groupoid associativity, if $g \in \Psi^*_{\text{EC}}(G)$, then, any $v \in V$ *s.t.* $g(u) = v$ would be constrained to allow to be acted by every h *s.t.* $(h, g) \in \mathcal{D}$. That is, unless partial transformations that we want to allow to be in each other domains are restricted to the same connected component of the graph, the supporting set of a partial EC* convolutions would still be bounded by the minimal degree.

2.4.4 Construction of path convolutions

To answer the limitation of partial convolutions, given $g \in \langle \mathcal{U} \rangle$ where $\mathcal{U} \subset \Psi^*_{\text{EC}}(G)$, the idea is to proceed with a foliation of g into pieces, each corresponding to an edge $e \in E$, and together generating another groupoid with a different associativity law, as follows.

Definition 83. Path groupoid

Let $\mathcal{U} \subset \Psi^*_{\text{EC}}(G)$. The *path groupoid* generated from $\mathcal{U}$, denoted $\mathcal{U} \ltimes G$, with composition rule $\mathcal{D}_\ltimes$, is the groupoid obtained inductively with:

1. $\mathcal{U} \ltimes_1 G = \{(g, v) \in \mathcal{U} \times V, v \in \mathcal{D}_g\} \subset \mathcal{U} \ltimes G$
2. $((g_n, v_n) \cdots (g_1, v_1), (h_m, u_m) \cdots (h_1, u_1)) \in \mathcal{D}_\ltimes \Leftrightarrow h_m(u_m) = v_1$
3. $((g_n, v_n) \cdots (g_1, v_1))^{-1} = (g_1^{-1}, g_1(v_1)) \cdots (g_n^{-1}, g_n(v_n))$

Call path its objects. Given a length $l \in \mathbb{N}$, denote $\mathcal{U} \ltimes_l G$ the subset composed of the paths that are the composition of exactly l paths of $\mathcal{U} \ltimes_1 G$.

Remark. This groupoid construction is inspired from the field of operator algebra where partial action groupoids have been extensively studied, *e.g.* Nica, 1994; Exel, 1998; Li, 2016.

Such groupoids usually come equipped with source and target maps. We also define the path map.

Definition 84. Source, target and path maps
Let a path groupoid $\mathcal{U} \ltimes G$. We define on it the *source map* α the *target map* β and the *path map* γ as:

$$\begin{cases} \alpha : (g_n, v_n) \cdots (g_1, v_1) \mapsto v_1 \in V \\ \beta : (g_n, v_n) \cdots (g_1, v_1) \mapsto g_n(v_n) \in V \\ \gamma : (g_n, v_n) \cdots (g_1, v_1) \mapsto g_n g_{n-1} \dots g_1 \in \Psi^*(V^0) \end{cases}$$

Remark. Note that the path groupoid can be seen as the local structure of the partial transformation groupoid.

Lemma 85.
Note the following properties:

1. $(p, q) \in \mathcal{D}_\ltimes \Leftrightarrow \alpha(p) = \beta(q)$
2. $\alpha(p) = \beta(p^{-1})$
3. $e_p^l = pp^{-1} = (\mathrm{Id}, \beta(p))$ and $e_p^r = p^{-1}p = (\mathrm{Id}, \alpha(p))$
4. γ is a groupoid partial action. We will denote γ_p instead of $\gamma(p)$.

Remark. Note that this time we won't use the notation $p(v)$ for $\gamma_p(v)$ for clarity.

One of the key object of our contruction is the use of φ-equivalence in order to transform a sum over a group(oid) of (partial) transformations, into a sum over the vertex set. With the current notion of path groupoid, searching for something similar amounts to searching for a graph traversal.

Definition 86. Traversal set
Let a graph $G = \langle V, E \rangle$ that is connected. A *traversal set* is a pair $(\mathcal{U}, \mathcal{T})$ of EC partial transformations subsets $\subset \Psi^*_{\mathrm{EC}}(G)$, such that

1. $\mathcal{U}$ is *edge-deterministic*, in the sense that an edge can only correspond to a unique g, *i.e.* $\forall g, h \in \mathcal{U} : \exists v \in V, g(v) = h(v) \Rightarrow g = h$

2. The EC partial transformations of $\mathcal{T}$ are restrictions of those of $\mathcal{U}$, *i.e.* $\forall g \in \mathcal{U}, \exists! h \in \mathcal{T}, \begin{cases} \mathcal{D}_h \subset \mathcal{D}_g \\ \forall v \in \mathcal{D}_h, h(v) = g(v) \end{cases}$
(equivalently, $\mathcal{T} \ltimes G$ is a path subgroupoid of $\mathcal{U} \ltimes G$ *s.t.* $|\mathcal{T}| = |\mathcal{U}|$)

3. The subgraph $G_{\mathcal{T}} = \langle V, \mathcal{T} \ltimes_1 G \rangle$ is a spanning tree of G.

We denote $(\mathcal{U}, \mathcal{T}) \in \mathrm{trav}(G)$, and denote by r the root of $G_{\mathcal{T}}$.
For $p \in \mathcal{T} \ltimes G \subset \mathcal{U} \ltimes G$, we denote $\gamma_p^{\mathcal{T} \ltimes G}$ and $\gamma_p^{\mathcal{U} \ltimes G}$ its path maps.

Remark. The assumption that the graph G is connected does not lose generality as the construction can be replicated to each connected component in the general case.

A traversal set $(\mathcal{U}, \mathcal{T})$ defines a φ-equivalence between the α-fiber of the root r and the vertex set V as follows.

Lemma 87. Path φ-Equivalence
Let $(\mathcal{U}, \mathcal{T}) \in \mathrm{trav}(G)$. Given $v \in V$, there exists a unique $p_v \in \mathcal{T} \ltimes G$ such that $\alpha(p_v) = r$ and $\beta(p_v) = v$. Denote $\mathcal{T} \ltimes^r G = \alpha^{-1}_{\mathcal{T} \ltimes G}\{r\}$. We can do the following construction:

1. Define $\varphi : p_v \mapsto v$.

2. Define $(p_v, p_u) \mapsto p_v^u \in \mathcal{U}^0 \ltimes^r G$ such that the sequence of partial transformations of p_v^u and p_v are the same (*i.e.* $\gamma^{\mathcal{U}^0 \ltimes G}_{p_v^u} = \gamma^{\mathcal{U} \ltimes G}_{p_v}$), and the source of p_v^u is the target of p_u (*i.e.* $\alpha(p_v^u) = \beta(p_u) = u$)

3. Define the external composition $p_v p_u = p_v^u p_u \in \mathcal{U}^0 \ltimes^r G$.

Then $\varphi : \alpha^{-1}_{\mathcal{T} \ltimes G}\{r\} \to V$ is a bijective partial equivariant map.

TODO: fix domain of φ and bijectivity
TODO: Vincent peux-tu m'aider à corriger lemme 82 et definition 83 pour que ca ait du sens ?

Proof. Bijectivity is a consequence of the spanning tree structure of $\mathcal{T}$. Equivariance because $\gamma_{p_v}(u) = \gamma_{p_v}\gamma_{p_u}(r) = \gamma_{p_v p_u}(r) = \varphi(p_v p_u)$. □

We can now define the convolution that is based on a path groupoid.

Definition 88. Path convolution
Let $(\mathcal{U}, \mathcal{T}) \in \mathrm{trav}(G)$. The *path convolution* is the partial convolution based on the path subgroupoid $\mathcal{T} \ltimes G$, which uses the groupoid partial action $\gamma := \gamma^{\mathcal{U}^0 \ltimes G}$ of the embedding groupoid zero-closure $\mathcal{U}^0 \ltimes G$.

(i) In what follows are the three expressions of the path φ-convolution

for signals $s_1, s_2 \in \mathcal{S}(V)$, and $u \in V$:

$$\begin{aligned}(s *_\varphi w) &= \sum_{v \in V} s[v]\, \gamma_{p_v}(w) \\ &= \sum_{\substack{p \in \mathcal{T} \ltimes G \\ s.t.\ \alpha(p)=r}} s[\varphi(p)]\, \gamma_p(w) \\ (s *_\varphi w)[u] &= \sum_{\substack{(a,b) \in V \\ s.t.\ \gamma_{pa}(b)=u}} s[a]\, w[b]\end{aligned}$$

(ii) The mixed formulations with $w \in \mathcal{S}(\mathcal{T} \ltimes G)$ are:

$$\begin{aligned}(w *_{\text{M}} s) &= \sum_{\substack{p \in \mathcal{T} \ltimes G \\ s.t.\ \alpha(p)=r}} w[p]\, \gamma_p(s) \\ (w *_{\text{M}} s)[u] &= \sum_{\substack{(p,v) \in \mathcal{T} \ltimes G \times V \\ s.t.\ \alpha(p)=r \\ s.t.\ \gamma_p(v)=u}} w[p]\, s[v]\end{aligned}$$

Remark. The role of $\mathcal{T}$ is to provide a φ-equivalence. The role of $\mathcal{U}$ is to extend every partial transformation $\gamma_g^{\mathcal{T} \ltimes G}$ to the domain of its unrestricted counterpart $\gamma_g^{\mathcal{U} \ltimes G}$.

Theorem 82 also holds for path groupoids, except that the domain-symmetric condition of 2.(i) is not needed.

Theorem 89. Characterization by equivariance to $\mathcal{U} \ltimes G$'s action
Let $(\mathcal{U}, \mathcal{T}) \in \mathrm{trav}(G)$.

(i) The class of linear transformations of $\mathcal{S}(V)$ that are equivariant to the path actions of $\mathcal{U} \ltimes G$ is exactly the path φ-convolution right-operators;

(ii) in the abelian case, they are also exactly the M-convolution left-operators.

Proof. Instead of the domain-symmetric condition that was used in the proof of the converse of Theorem 82 (2.(i)), we use the fact that any vertex can be reached with an action from the root of the spanning tree of the traversal set. Indeed, given $v \in V$, as we have $\gamma_{p_v}(r) = v$, then $\gamma_{p_v}(\delta_r) = \delta_v$. Therefore, by developping a linear transformation $f(s)$ on the dirac family, and commuting f with γ_{p_v}, we obtain that $f(s) = s *_\varphi w$, where $w = f(\delta_r)$. The rest of the proof is similar to that of Theorem 82. □

Remark. Note that $\mathcal{U} \ltimes V$'s action is almost the same as the groupoid partial action of $\Upsilon = \langle \mathcal{U} \rangle$ (only "almost" because not all combinations of partial transformations might exist in the paths). However $\mathcal{U} \ltimes V$ alleviates the limitations of Υ we discussed earlier.

Edge convolution operators

The counterparts of EC* convolution operators for path convolutions, are indeed path convolution operators obtained with supporting set $\mathcal{N} \subset \mathcal{T} \ltimes_1 G$ which any graph can admit. By extrapolation, we can coin them *edge convolution operators*. As shown by this section, to construct one, all we need is a traversal set of partial transformations $(\mathcal{U}, \mathcal{T})$.

2.5 Conclusion

In this chapter, we constructed the convolution on graph domains.

1. We first saw that classical convolutions are in fact the class of linear transformations of the signal space that are equivariant to translations. For signals defined on graph domains, there is no natural definition of translations.

2. Therefore, we adopted a more abstract standpoint and considered in the first place any kind of transformation of the vertex set V. Hence, given a subgroup of transformation Γ, we constructed the class of linear transformations of the signal space that are equivariant to it. This provided us with an expression of a convolution based on this subgroup, and a bijective equivariant map between Γ and V, in order to transport a sum over Γ into a sum over V. We also proposed a simpler expression in the abelian case.

3. Then, we introduced the role of the edge set E, and we constrained Γ by it. This allows us to obtain a characterization of admissibility of convolutions by Cayley subgraph isomorphism, and to analyze intrinsic properties of the constructed convolution operator, namely locality and weight sharing. We also discussed operators with a smaller kernel, in particular those that are EC*, as they are simpler to construct.

4. Finally, we overcame the limitation that some graphs only have trivials or low degree Cayley subgraphs. In this case, we rebased our construction on groupoids of partial transformations Υ as a first iteration, but this one didn't overcome fully the above-mentioned limitation. As a last iteration, we broke down the previous construction into elementary partial actions onto the edges, recomposed into

path groupoids $\mathcal{U} \ltimes G$. Similarly, equivariance characterization and intrinsic properties hold, and the simpler EC* construction is also possible.

Summary of practical EC* convolution operators

3. For graphs that are quite regular, in the sense that they contain an above-low-degree Cayley subgraph (degree $d \geq 4$), we saw in Section 2.3.3 that all we need to construct an EC* convolution operator is a generating set $\mathcal{U}$ of transformations, without the need of composing its elements, and optionally (in the non-abelian case) to move a local patch $\mathcal{K}_{\mathrm{Id}}$ over the graph domain.

4. For a general graph, we saw in Section 2.4.4 that all we need to construct an EC* path convolution operator is a traversal set $(\mathcal{U}, \mathcal{T})$ of partial transformations, without the need to compose the paths.

In the next chapter, we will encounter examples of EC and EC* convolution operators defined on graphs, that can be expressed under group representations or under path groupoid representations.

Chapter 3
Deep learning on graph domains

Contents

Chapter overview

Our goal in this chapter is to understand how neural networks can be extended to other domains than what they were intended for, and in particular to graph domains. To this end, in Section 3.1, we make an effort to interpret the linear algebra that underpins a layer to build our intuition. First we state the obvious by explaining further the interpretation of tensor space as a neural space, as well as how to juggle between tensors and signals. Then, we propose a representation based on graphs. Between two layers, a propagation graph explains how the propagation is done. On the input layer, the neurons can have an underlying graph structure. We show a relation between these two graphs, obtained if and only if the local receptive fields of the neurons are intertwined. By introducing the notion of weight sharing in our analysis, we discover that a layer on any domain can be expressed by a linear ternary operation, that we call *neural contraction*. Its operands are the *input signal* X, the *weight kernel* Θ, and the *weight sharing scheme* S. We denote $\widehat{\Theta SX}$. In Section 3.2, we study this ternary representation. We see that it is generic. We show that under sparse priors, it can be implemented efficiently better than with sparse classes of common libraries by adopting an approach based on local receptive fields. With an experiment based on the ternary representation, we see how exploiting symmetries is beneficial, which justifies the use of convolutions. In Section 3.3, we study the effect of learning how the weights are shared, which amounts to learn both S and W. We explore this avenue for graph domains, with experiments on grid and covariance graphs. Finally, in Section 3.4, we investigate an exemple of a CNN architecture used for graph signals. The convolution is based on translations on graphs which define the weight sharing scheme S of the convolutional layer. We present the model of translations and the approximation we use, the subsampling layer, the data augmentation, and experiments.

3.1 Layer representations

Let $\mathcal{L} = (g, h)$ be a neural network layer, where $g : I \to O$ is its linear part, $h : O \to O$ is its activation function, I and O are its input and output spaces, which are tensor spaces.

3.1.1 Neural interpretation of tensor spaces

Recall from Definition 1 that a tensor space has been defined such that its canonical basis is a Cartesian product of canonical bases of vector spaces. Let $I = \bigotimes_{k=1}^{p} \mathbb{V}_k$ and $O = \bigotimes_{l=1}^{q} \mathbb{U}_l$. Their canonical bases are denoted $\mathbb{v}_k = (\mathbb{v}_k^1, \ldots, \mathbb{v}_k^{n_k})$ and $\mathbb{u}_l = (\mathbb{u}_l^1, \ldots, \mathbb{u}_l^{n_l})$.

Remark. Note that a tensor space is isomorph to the signal space defined over its canonical basis.

More precisely, we have the following relation.

Lemma 90. Relation between tensor and signal spaces
Let $\mathbb{V}$, $\mathbb{U}$ be vector spaces, and $\mathbb{v}$, $\mathbb{u}$ be their canonical bases. Let $\mathbb{T}$ be a tensor space. $\otimes$ and $\times$ denote tensor and Cartesian products. Then,

(i) $\mathbb{V} \cong \mathcal{S}(\mathbb{v})$

(ii) $\mathbb{V} \otimes \mathbb{U} \cong \mathcal{S}(\mathbb{v} \times \mathbb{u})$

(iii) $\mathbb{V} \otimes \mathbb{T} \cong \mathcal{S}_{\mathbb{T}}(\mathbb{v})$

where $\mathcal{S}_{\mathbb{T}}$ are signals taking values in $\mathbb{T}$ (and $\mathcal{S}$ are real-valued signals).

Proof. (i) Given $x \in \mathbb{V}$, define $\widetilde{x} \in \mathcal{S}(\mathbb{v})$ such that $\forall i, \widetilde{x}[\mathbb{v}^i] = x[i]$. The mapping $x \mapsto \widetilde{x}$ is a linear isomorphism.

(ii) $\widetilde{x}[\mathbb{v}^i, \mathbb{u}^j] = x[i, j]$

(iii) $\widetilde{x}[\mathbb{v}^i] = x[i, :, \ldots, :]$

□

Let $d \leq n_k$ and $e \leq n_l$. Define V and U as the Cartesian products $V = \bigtimes_{k=1}^{d} \mathbb{v}_k$ and $U = \bigtimes_{l=1}^{e} \mathbb{u}_l$. Thanks to Lemma 90, we can identify the input and output spaces as $I = \mathcal{S}(V) \otimes \bigotimes_{k=d+1}^{p} \mathbb{V}_k$ and $O = \mathcal{S}(U) \otimes \bigotimes_{l=e+1}^{q} \mathbb{U}_l$. As $\mathcal{S}(\mathbb{v}) \otimes \mathbb{T} = \mathcal{S}_{\mathbb{T}}(\mathbb{v})$, an object of V or U can be interpreted as the representation of a *neuron* which can take multiple values.

In what follows, without loss of generality, we will make the simplification that a neuron can only take a single value (we do not consider input channels and feature maps yet). We'll thus consider that $I = \mathcal{S}(V)$ and $O = \mathcal{S}(U)$, where V is the set of *input neurons*, and U is the set of *output neurons*.

3.1.2 Propagational interpretation

Let $\mathcal{L} = (g, h)$, recall that $g : \mathcal{S}(V) \to \mathcal{S}(U)$ is characterized by a connectivity matrix W such that, $g(x) = Wx$.

Remark. Using the mapping defined in the proof of Lemma 90, for notational conveniency, we'll abusively consider x as a vector (enventually reshaped from a tensor), and W as an object of a binary tensor product for its indexing (*i.e.* $W[u, v] := W[i, j]$ where $u = \mathbb{u}^i$ and $v = \mathbb{v}^j$).

Definition 91. Propagation graph
The *propagation graph* $P = \langle (V, U), E_P \rangle$ of a layer $\mathcal{L} = (W, h)$ is the bipartite graph that has the connectivity matrix W for bipartite adjacency matrix.

The propagation graph defines an input topological space $\mathcal{T}_V$, and an output topological space $\mathcal{T}_U$.

Definition 92. Topological space
A *topological space* is a pair $\mathcal{T} = (X, \mathcal{O})$, where X is a set of points, $\mathcal{O}$ is a set of sets that is closed under intersection (the *open sets*), and such that every point $x \in X$ is associated with a set $\mathcal{N}(x) \in \mathcal{O}$, called its *neighborhood*.

Hence, the *neural topologies* $\mathcal{T}_V$ and $\mathcal{T}_U$ are defined as

1. $\mathcal{T}_V = (V, \mathcal{O}(U))$, with $\forall v \in V, \mathcal{N}(v) = \{u \in U, v \overset{P}{\sim} u\}$
2. $\mathcal{T}_U = (U, \mathcal{O}(V))$, with $\forall u \in U, \mathcal{N}(u) = \{v \in V, v \overset{P}{\sim} u\}$

In particular, given an output neuron $u \in U$, a neighborhood $\mathcal{N}(u)$ is also called a *local receptive field* (LRF), that we denote $\mathcal{R}(u)$.

3.1.3 Graph representation of the input space

Let's consider that the input neurons V have a (possibly edge-less) graph structure $G = \langle V, E \rangle$.

Definition 93. Underlying graph structure
An underlying graph structure of an input space V is simply a graph such that V is its vertex set.

For our definition of edge-constrained layer, we want that the LRF are constrained by the edges of the underlying graph. So we define it as follows.

Definition 94. Edge-constrained layer
A layer $\mathcal{L} : G = \langle V, E \rangle \to U$, is said to be *edge-constrained* (EC) if:

1. There is a one-to-one correspondence $\pi : V_\pi \to U$, where $V_\pi \subset V$.
2. Given an output neuron u, an input neuron v is in its receptive field, if and only if, v and the π-fiber of u are connected in G,
$$\textit{i.e. } \forall u \in U, v \in \mathcal{R}(u) \Leftrightarrow v \overset{E}{\sim} \pi^{-1}(u)$$

Figure 11 illustrate an example of a local receptive field of an edge-constrained layer.

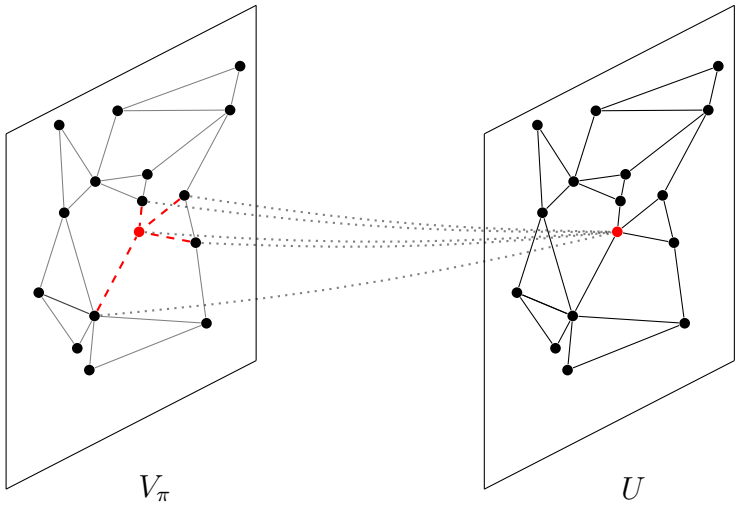


Figure 11: An LRF of a vertex (in red) of an EC layer

Remark. Note that EC convolutions from Chapter 2 are indeed EC layers.

Also, the following property is important.

Proposition 95. Let W the connectivity matrix of an EC layer *w.r.t.* a graph $G = \langle V, E \rangle$, which adjacency matrix is A. Then W is *masked by* A, *i.e.*

$$A[i,j] = 0 \Rightarrow W[i,j] = 0$$

Proof. Suppose $A[i,j] = 0$, then $v_i \nsim v_j$, hence $v_i \notin \mathcal{R}(\pi(v_j))$ so $W[i,j] = 0$. □

We have the following characterization to determine which layers admit an underlying graph for which they are EC layers.

Proposition 96. EC Characterization with receptive fields
Let a layer $\mathcal{L} : V \to U$, $V_\pi \subset V$, and a one-to-one correspondence $\pi : V_\pi \to U$. There exists a graph $G = \langle V, E \rangle$ for which $\mathcal{L}$ is EC, if and only if, the receptive fields are *intertwined* (*i.e.* $\forall a, b \in V_\pi, a \in \mathcal{R}(\pi(b)) \Leftrightarrow b \in \mathcal{R}(\pi(a))$).

Proof. ($\Rightarrow$) Thanks to $a \in \mathcal{R}(\pi(b)) \Leftrightarrow a \overset{E}{\sim} b \Leftrightarrow b \in \mathcal{R}(\pi(a))$

($\Leftarrow$) If the receptive fields are intertwined, then the relation defined as $a \sim b \Leftrightarrow a \in \mathcal{R}(\pi(b))$ is symmetric, and thus can define an edge set.

□

Therefore, any layer that has its receptive fields intertwined, admits an *underlying* graph structure. For example, a 2-d convolution operator can be rewritten as an EC* convolution on a lattice graph, and as an EC convolution on a grid graph.

3.1.4 Novel ternary representation with weight sharing

Weight sharing refers to the fact that some parameters of the connectivity matrix W are equal, and stay equal after each learning iteration. In other words they are tied together. From a propagational point of view, this amounts to label the edges of the propagation graph P with weights, where weights can be used multiple times to label these edges. Supposed W is of shape $m \times n$ and there are ω weights in the kernel used to label the edges. Given a input neuron i and an output neuron j, the edge labelling can be expressed as:

$$W[j, i] = \theta[h] = \theta^T a \tag{48}$$

where θ, the weight kernel, is a vector of size ω and a is a one-hot vector (full of 0s except for one 1) of size ω with a 1 at the index h corresponding

to the weight that labels the edge $i \sim j$; or a is the *zero* vector in case $i \nsim j$.
This equation (48) can be rewritten as a tensor contraction under Einstein summation convention, by noticing that a depends on i, j and by defining a tensor S such that $a = S[:, i, j]$, as follows:

$$W_{ij} = \theta_k S^k{}_{ij} \tag{49}$$

Therefore, the linear part of $\mathcal{L}$ can be rewritten as:

$$g(x)_j = \theta_k S^k{}_j{}^i x_i \tag{50}$$

If we consider that the layer $\mathcal{L}$ is duplicated with input channels and feature maps, then θ, x and $g(x)$ become tensors, denoted Θ, X and $g(X)$. Usually, for stochastic gradient descent, X and $g(X)$ are also expanded with a further rank corresponding to the batch size and we obtain:

$$g(X) = \Theta S X \text{ where } \begin{cases} W_{pq}{}^{ij} = \Theta_{pq}{}^k S_k{}^{ij} \\ g(X)_{jq}{}^b = W_{jq}{}^{ip} X_{ip}{}^b \end{cases} \tag{51}$$

index	size	description
i	n	input neuron
j	m	output neuron
p	N	input channel
q	M	feature map
k	ω	kernel weight
b	B	batch instance

Table 2: Table of indices

Since the multiplicative expression $\Theta S X$ is written regardless of the ordering of the tensors ranks and is defined by index juggling, we will write instead $\widehat{\Theta S X}$ to avoid confusion. $\widehat{\cdots}$ is a ternary operator which

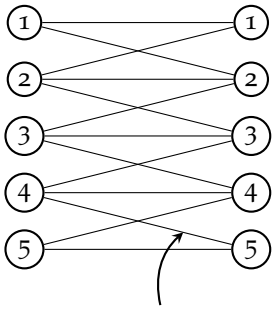
we will call *neural contraction*. Note that it is associative and commutative. This can be seen by the index symmetry of (52), which rewrites (51), and where the sum symbols Σ and scalar values commute:

$$\widehat{\Theta SX}[j,q,b] = \sum_{k=1}^{\omega} \sum_{p=1}^{P} \sum_{i=1}^{n} \Theta[k,p,q]\, S[k,i,j]\, X[i,p,b] \tag{52}$$

Definition 97. Ternary representation, neural contraction
The *ternary representation* of a layer $\mathcal{L} : X \mapsto Y$, with activation function h, is the equation $Y = h\left(\widehat{\Theta SX}\right)$, where the *neural contraction* $\widehat{\cdots}$ is defined by (52), Θ is the *weight kernel*, and S is called the *weight sharing scheme*.

Remark. In (48), we defined $a = S[:,i,j]$ as a one-hot vector when $i \sim j$, as its role is to select a weight in $\theta = \Theta[p,q,:]$. However, a can also do this selection *linearly*, so in fact it is not necessarily a one-hot vector.

Figure 12 depicts an example of how the equation (51) labels the edges of P.
The ternary representation uncouples the roles of Θ and S in W, and is the most general way of representing any kind of partially connected layer with weight sharing.



$$W[p, q, i = 4, j = 5] = \sum_{k=1}^{\omega} \Theta[p, q, k] S[k, 4, 5]$$

Figure 12: Example of a propagation graph P for a given input channel p and feature map q. The edge $4 \sim 5$ is labelled with a linear combination of kernel weights from $\Theta[p, q, :]$. In the usual case, $S[k, 4, 5]$ is a one-hot vector that selects a single kernel weight: $\exists h, W[p, q, 4, 5] = \Theta[p, q, h]$.

3.2 Study of the ternary representation

In this section, we study the ternary representation, which is the general representation with weight sharing we obtained above. We already saw that it is linear, associative and commutative.

3.2.1 Genericity

The ternary representation can represent any kind of layer. We explain below how to obtain standard ones. See Table 2 and Table 3 for a description of the indices, notations and shapes.

- ➢ To obtain a fully connected layer, one can choose $\omega = nm$ such that the slices $S[:, i, j]$ constitute every possible one-hot vectors.

- ➢ To obtain a convolutional layer, one can choose ω to be the size of the kernel. S would contain one-hot vectors. A stride > 1 can be

obtained by removing the corresponding dimensions. If the convolution is a classical convolution, or more generally, is characterized by a Cayley subgraph (see Chapter 2), then S would be circulant along the input neurons rank in the canonical basis.

- To obtain a graph convolutional layer (Kipf and Welling, 2016, see description in Section 1.3.5), one can choose $B = 1$, $\omega = 1$, and S (viewed as a matrix) to be the normalized adjacency matrix $\widetilde{A}$.

- Other variants of GCN can also be obtained, for instance to obtain Chebychev filters (Defferrard et al., 2016), S would be $\sum_{i=0}^{k} T_i(\widetilde{L})$.

- To obtain a graph attention layer (Velickovic et al., 2017, see description in Section 1.3.5), one can concatenate K graph convolutional layers, where K is the number of attention heads, and with $\widetilde{A}$ filled with the learned attention coefficients. Instead of concatenation, one could also choose $\omega = K$, and given an attention head k, the slices $S[k, :, :]$ would be matrices containing the coefficients.

- A topology-adaptative graph convolution layer (Du et al., 2017, see description in Section 1.3.5) is a neural contraction layer for which S contains the powers of $\widetilde{A}$ along the first rank.

- A mixture model convolutional layer (Monti et al., 2016, equations (9) and (10)) is a neural contraction layer for which S contains the values of the weighting functions *i.e.* $S[k, i, j] = w_k(u[i, j])$.

- A generalized convolution (Vialatte et al., 2016, equations (1) and (10)) is a neural contraction layer for which S is the allocation tensor A^e which has the sparse priors mentioned in the next subsection.

- Any partially connected layer with (or without) weight sharing can be obtained with appropriate construction of S.

3.2.2 Sparse priors for the classification of signals

In the more general case, the scheme S is a real-valued tensor, which may introduce more weights in the layer. However, for Euclidean convolutions, S is already determined by the Euclidean convolution operator and does not count for more memory. To imitate this property, we can focus on the class of layer with these sparse priors:

1. Given a couple of neuron indices (i, j), $S[:, i, j]$ is either a one-hot vector, or the zero vector, as in (48).

2. Given an output neuron index j, a weight $\theta[h]$ can appear only once in its LRF.

Remark. For these sparse priors we consider the case of classification of signals. In the case of semi-supervised node classification, S would only be sparse between the rank indexed by i and the one indexed by j, but not on the other rank.

Therefore, we can describe S with a *weight assignment table* T, which is a matrix such that

$$T[i, j] = \begin{cases} h \text{ if } \exists! h, \theta^T S[:, i, j] = \theta[h] \\ 0 \text{ else} \end{cases} \tag{53}$$

In turn, T can be described by the set of transformations (or partial transformations), defined as

$$g_h(j) = i \Leftrightarrow T[i, j] = h \tag{54}$$

where each (partial) transformation g_h corresponds to the weight indexed by h.

Without loss of generality, let us reduce to the simple case where $B = P = Q = 1$. The neural contraction (52) rewrites as:

$$\widehat{\theta S x}[j] = \sum_{k=1}^{\omega} \sum_{i=1}^{n} \theta[k]\, S[k,i,j]\, x[i] \tag{55}$$

$$= \sum_{h=1}^{\omega} \theta[h]\, x[g_h(j)] \tag{56}$$

which for EC layers, can be rewritten as a convolution of graph signals as studied in Chapter 2, since the (partial) transformations are invertible thanks to the sparse priors. Conversely, given a convolution of graph signals (formulated as $*_\varphi$ or $*_{\text{M}}$), we can derive an assignment table T and its corresponding scheme S to obtain a neural contraction.

3.2.3 Efficient implementation under sparse priors

Remark. This section applies in supervised settings but not in semi-supervised setting for which it is best to use sparse functions of common deep learning libraries.

What is the fastest way to compute $\widehat{\Theta S X}$?

As the equation (51) is associative and commutative, there are three ways to start to calculate it: with ΘS, SX, or ΘX, which we will call *middle-stage tensors*. The computation of a middle-stage tensor is the bottleneck to compute (51) as it imposes to compute significantly more entries than for the second tensor contraction. In Table 3, we compare their shapes. We refer the reader to Table 2 for the denomination of the indices.

tensor	shape
Θ	$\omega \times N \times M$
S	$\omega \times n \times m$
X	$n \times N \times B$
ΘS	$n \times m \times N \times M$
SX	$\omega \times m \times N \times B$
ΘX	$\omega \times n \times M \times B$
$\widehat{\Theta S X}$	$m \times M \times B$

Table 3: Table of shapes

We usually want to have $\omega \ll n$ and $\omega \ll m$, which means that we have weight kernels of small sizes (for example in the case of images, convolutional kernel are of size significantly smaller than that of the images). Also, the number of input channels N and of feature maps M are roughly in the same order, with $N < M$ more often than the contrary. So in practice, the size of ΘS is significantly bigger than the size of SX and of ΘX, and the size of SX is usually the smallest.

How to exploit S sparsity ?

Also, in ususal supervised settings, S is sparse as $S[:, i, j]$ are one-hot vectors. So computing SX should be faster than computing ΘX, provided we exploit the sparsity. Although S is very sparse as it contains at most a fraction $\frac{1}{w}$-th of non-zero values, it is only sparse along the first rank, which makes implementation with sparse classes of common deep learning libraries use less parallelization than it can. However, we can benefit from the sparse priors mentioned in Section 3.2.2.

So we proceed differently. The idea is to use a non-sparse tensor X_{LRF} that has a rank that indexes the LRF, and another rank that indexes elements of these LRF, in order to lower the computation to a dense matrix multiplication (or a dense tensor contraction) which is already well optimized. This approach is similar to that proposed in Chellapilla et al., 2006, which

is also exploited in the cudnn primitives (Chetlur et al., 2014) to efficiently implement the classical convolution.

The LRF representation

In our case, it turns out that X_{LRF} can be exactly SX, as given fixed b, p, and j, $SX[:, j, p, b]$ corresponds to entries of the input signal $X[:, p, b]$ restrained to a LRF $\mathcal{R}_j$ of size ω. Therefore,

$$\exists\, \textsc{lrf}_j = [i_1, \ldots, i_\omega] \text{ s.t. } SX[:, j, p, b] = X[\textsc{lrf}_j, p, b] \tag{57}$$

The elements of $\textsc{lrf}_j$ can be found by doing a lookup in the one-hot vectors of S, provided each kernel weight occurs exactly once in each LRF. We have:

$$R_j[k] = i_k \text{ s.t. } S[:, i_k, j][k] = 1 \tag{58}$$

This lookup needs not be computed each time and can be done beforehand. Finally, if we define $\textsc{lrf} = [\textsc{lrf}_1, \ldots, \textsc{lrf}_m]$, (57) gives:

$$SX = X[\textsc{lrf}, :, :] \tag{59}$$

The equation (59) is computed with only $\omega \times m$ assignations and can be simply implemented with automatic differentiation in commonly used deep learning libraries.

Benchmarks

To support our theoretical analysis, we benchmark three methods for computing the tensor contraction SX:

- naively using dense multiplication,
- using sparse classes of deep learning libraries,
- using the LRF based method we described above.

We run the benchmarks under the sparse priors mentioned in Section 3.2.2. For each method, we make 100 runs of computations of SX, with S and X being randomly generated according to the assumptions. In Table 4, we report the mean times. The values of the hyperparameters were each time $n = m = N = M = B = 256$, and $\omega = 4$. The computations were done on graphical processing units (GPU)[1].

Method	Mean time
Naive	950 ms
Sparse	413 ms
LRF	**365** ms

Table 4: Benchmark (100 runs each)

We observe that the LRF method is faster, since the sparse implementation isn't optimized for this use case.

3.2.4 Influence of symmetries

In the case of images, or other signals over a grid, the grid structure of the domain defines the weight sharing scheme S of the convolution operation. For example, for a layer $\mathcal{L} : X \mapsto Y = h(\widehat{\Theta SX})$, and given fixed b, p, q, the classical convolution can be rewritten as

$$y[j] = h\left(\sum_{k=1}^{\omega} \theta[k] \sum_{i=1}^{n} S[k,i,j]\, x[i]\right) \tag{60}$$

$$= h\left(\sum_{k=1}^{\omega} \theta[k]\, x_{\text{LRF}}[k,j]\right) \tag{61}$$

Where $x_{\text{LRF}}[k,j]$ can be obtained by matching (61) with the expression given by the definition (see Definition 36). So, $S[k,:,j]$ is a one-hot vector

[1]The GPU was an Nvidia Geforce GTX1060, the CPU was an Intel Xeon E5-1630v4 at 3.70GHz with 4 cores. We used tensorflow and the function gather_nd for the assignations.

that specifies which input neuron in the LRF of y is associated with the k-th kernel weight, and the index of the 1 is determined by $x_{\text{LRF}}[k, j]$.
That is to say, in the ternary representation, Θ captures the information the layer has learnt, whereas S captures how the layer exploits the symmetries of the input domain.

Visual construction

Visually, constructing S amounts to move a rectangular grid over the pixel domain, as depicted on Figure 13 where each point represents the center of a pixel, and the moving rectangle represents the LRF of its center j. Each of its squares represents a kernel weight $\theta[k]$ which are associated with the pixel $x_{\text{LRF}}[k, j]$ that falls in it.

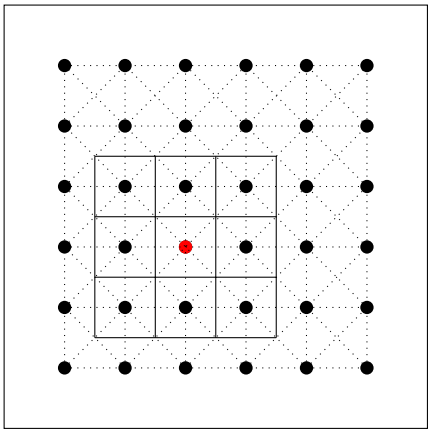


Figure 13: Weight assignement of convolutions on pixel domains

The case of images is very regular, in the sense that every pixels are regularly spaced out, so that obtained S is circulant along its last two ranks. This is a consequence of the translational symmetries of the input domain, which underly the definition of the convolution, as seen in Chapter 2.

- What happens if we loose these symmetries?

To answer this question, we make the following experiment (Vialatte et al., 2016):

1. We distort the domain by moving the pixels randomly. The radial displacement is uniformly random with the angle, and its radius follows a gaussian distribution $\mathcal{N}(0, \sigma)$.

2. Then we compare performances of shallow CNNs expressed under the ternary representation, for which S is constructed similarly than with the above visual construction, for different values of σ.

The visual construction of S on distorded domain is depicted by Figure 14.

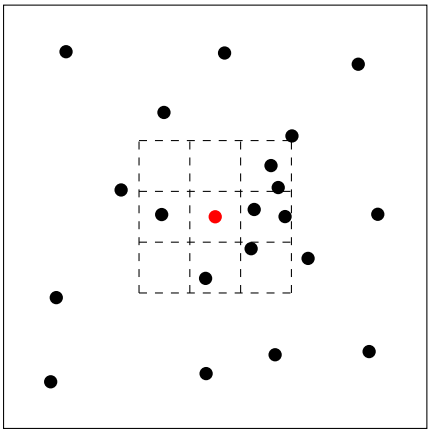


Figure 14: Weight assignement in generalized convolution on distorded domains

We run a classification task with standard hyperparameters on a toy dataset (we used MNIST, LeCun et al., 1998). The results are reported in Figure 15.
The bigger is σ, the less accurate are the symmetries of the input domain, up to a point where the ternary representation becomes almost equivalent to a dense layer. The results illustrate nicely this evolution, and stress out the importance of trying to leverage symmetries when defining new convolutions.
Moreover, they indicate that the ternary representation also allows to improve performances compared to using dense layers, providing we are

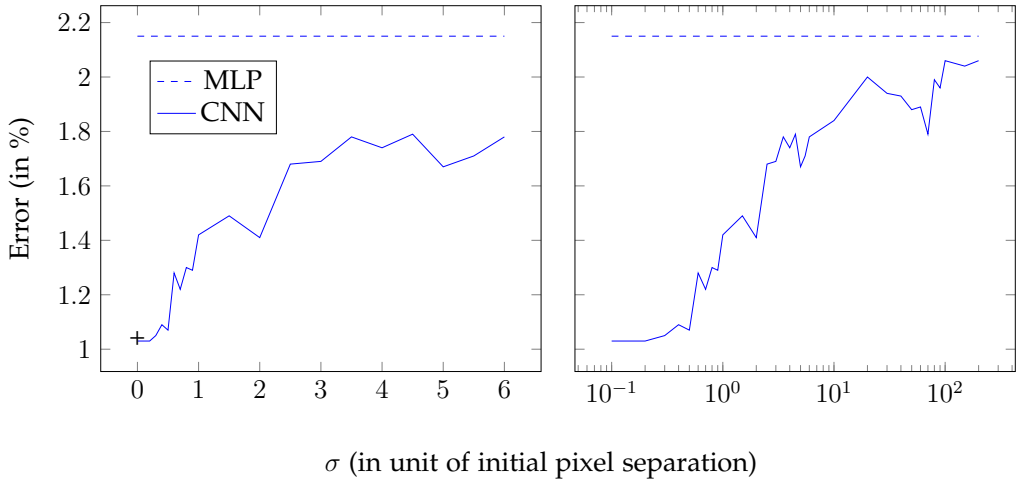


Figure 15: Error in function of the standard deviation σ, for generalized CNNs and an MLP, each with 500 weights.

able to create a relevant weight sharing scheme S to exploit symmetries. For example, in the case the underlying graph structure of the input is embedded in a 2-d Euclidean space, S can be created just as in this experiment.

3.2.5 Experiments with general graphs

The neural contraction is not usable off-the-shelf for general graphs since the weight sharing scheme S needs to be specified. One strategy is to learn it alongside the weight kernel Θ, see Section 3.3. Another one consists in inferring it from the graph itself, see Section 3.4. But first let us try some ideas based on randomizations.

Supervised classification of graph signals

We test the idea to fill the schemes S with one-hot vectors randomly, but with respect to the sparse priors mentioned in Section 3.2.2. We apply neural contractions in a depthwise separable fashion similarly than for depthwise separable convolutions (Chollet, 2016). That is, we first propa-

gate the neurons values on the propagation graph for each of the N input feature maps independently. In the process, we use one different realization of S per input feature map. Then we apply a fully connected layer in the feature space (which is a convolution with trivial sliding window). Since this amounts to do M (one for each output feature map) learned weighted averages of N random realizations, we call that a Monte-Carlo (MC) module. We call *Monte-Carlo Networks* (MCNet) the corresponding neural networks.

We replicated the experimental setup done by Defferrard et al., 2016 on the 20NEWS dataset. We refer the reader to Section 1.3.4 for the preliminary discussion on this experiment. We used the architecture depicted on Figure 16. It starts with 2 MC modules with 64 output feature maps, followed by a trivial convolution with 1 output feature map, then a FC layer with 500 hiddden units and a final FC layer for classification. A residual connection is added between the input and the first FC layer. For the MC modules, we used the graph connecting 2 nearest neighbours. Therefore this architecture can be seen as a small random perturbation of the MLP from Table 1. Each layer except for the last is followed by 20% dropout. As usual, we use reLU activations except for the last that has softmax activation.

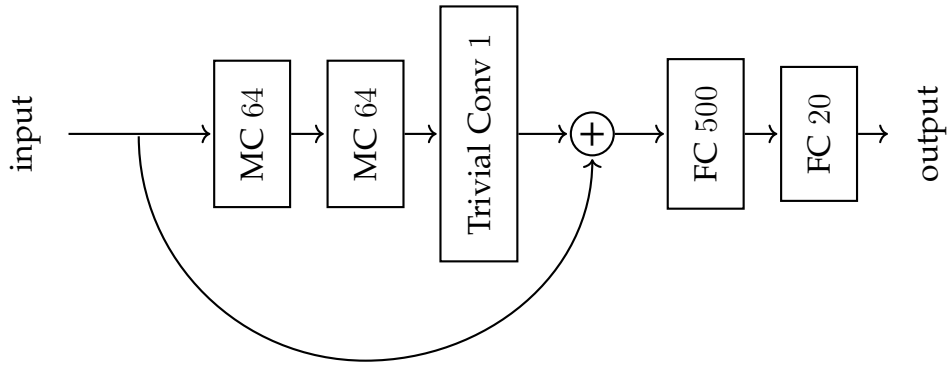


Figure 16: Diagram of the MCNet architecture used

After 100 runs of 10 epochs, MCNet obtained a mean accuracy of $70.74 \pm 2.19\%$,

which is a gain that is statistically significant compared to ChebNet (65.76%, see Table 1), but not against the former MLP ($71.46 \pm 0.08\%$). It is to be noted that the variance of MCNet is quite high and cherry picking the max instead of considering the mean improves the result to 72.62%, which is better than the MLP's max at 72.25%. Therefore, in some random realizations, MCNet is capable of leveraging the underlying graph structure (and being state-of-the-art).
This is the least to be expected. But when the quality of the graph is low, there may not be anything better to demonstrate. A dataset with a graph that suits well the underpining structure of the domain is often resembling a grid graph to some extent. Therefore we expect the model that will be presented in Section 3.4 to be more relevant in most cases.

Semi-supervised classification of nodes
A similar kind of randomization based idea that come in mind when transposing the question to the semi-supervised task is to use dropout on the edges of the graph. We call that *graph dropout*. We test this approach on citation networks, for which we set $\omega = 1$ (*i.e.* one graph). The obtained architecture amounts to a GCN (Kipf and Welling, 2016) to which graph dropout is applied. We report our results in Section 3.3.6, Table 8, where we compare them with other models.

3.3 Learning the weight sharing scheme

3.3.1 Discussion

In the ternary representation $Y = h(\widehat{\Theta S X})$ of a layer $\mathcal{L}$, the weight kernel Θ is usually the only operand that is learned, and the role of S is to label the edges of the propagation graph P with these weights. Recall that, as noted after Definition 97, S needs not be sparse and composed of one-hot vectors. In that case, the labelling is done linearly as depicted previously in Figure 12. Therefore, the weight sharing scheme S can also be updated during the learning phase. This can be interpreted as learning a convolution-like operator on the underlying graph G (Vialatte et al., 2017).

Remark. When S does not have the sparse priors mentioned in Section 3.2.2, we must not have more parameters in S and Θ than in W, so that the weight sharing still makes sense. If we call l the number of edges in the resulting propagation graph P, then the former assumption requires $l\omega + \omega NM \leq lNM$ or equivalently $\frac{1}{\omega} \geq \frac{1}{NM} + \frac{1}{l}$. It implies that the number of weights per filter ω must be lower than the total number of filters NM and than the number of edges l, which is always the case in practice.

3.3.2 Experimental settings

In our experiments, we learn the scheme S and the kernel Θ simultaneously.

Constraints for supervised classification of graph signals

Because of our inspiration from CNNs, we propose constraints on the parameters of S. Namely, we impose them to be between 0 and 1, and to sum

to 1 along the first rank (the rank that is contracted in the product ΘS).

$$\forall (k,i,j),\ S[k,i,j] \in [0,1] \tag{62}$$

$$\forall (i,j),\ \sum_{k=1}^{\omega} S[k,i,j] = 1 \tag{63}$$

Therefore, the vectors on the first rank of S can be interpreted as performing a positive weighted average of the parameters in Θ. Also, we choose to aim our study at graph signals. Hence, we consider a graph $G = \langle V, E \rangle$ and that the layer is EC. For this reason, Proposition 95 tells us that the connectivity matrix W of the layer is masked by the adjacency matrix A. Therefore, S is also masked by A *i.e.* we impose the constraint that

$$A[i,j] = 0 \Rightarrow \forall k, S[k,i,j] = 0 \tag{64}$$

Constraints for semi-supervised classification of nodes

For the case of citation networks, we consider only one graph and thus we set $\omega = 1$ so that S and Θ are matrices. Additionaly, we impose rows of the scheme S to be normalized so that each node receives a normalized information from its neighborhood. In our experiments, we choose to use softmax normalization because it is of standard usage (but other normalizations could do as well).

Name

In (Vialatte et al., 2017), we used the term *Local Receptive Graph layers*, since Proposition 96 relates G with local receptive fields defined by W. However it was cumbersome so in this manuscript we will just call them *Graph Contraction Layer* in reference to the neural contraction (see Definition 97). We call *Graph Contraction Networks* (GCT) the corresponding neural networks. The main addition to the paper (Vialatte et al., 2017) from Section 3.3 is Section 3.3.6.

Initialization

In supervised settings, we introduce three types of initialization for the scheme S. The last two have the sparse priors mentioned in Section 3.2.2:

1. uniform random: parameters of S are simply initialized with a uniform random distribution with limits as described by Glorot and Bengio, 2010.

2. random one-hot: one-hot vectors are distributed randomly on the $S[:, i, j]$, with the constraint that for each LRF, a particular one-hot vector can only be distributed at most once more than any other.

3. circulant one-hot: one-hot vectors are distributed in a circulant fashion on the $S[:, i, j]$, so that on Euclidean domains, the initial state of S correspond exactly to the weight sharing scheme of a standard convolution.

In semi-supervised settings, we initialize the scheme S using the normalized adjacency matrix $\widetilde{A}$ to which we add a gaussian noise $\mathcal{N}(0, \sigma)$, where the standard deviation σ is determined following Glorot and Bengio, 2010.

3.3.3 Experiments with grid graphs

We experiment GCTs on the MNIST dataset (see Section 1.3.3). We use a shallow architecture made of a single ternary layer with 50 feature maps, without pooling, followed by a FC layer of 300 neurons, 50% dropout, and terminated by a FC layer of 10 neurons with softmax activation. ReLu activations are used. Input layers are regularized by a factor weight of 10^{-5} (Ng, 2004). We optimize with ADAM (Kingma and Ba, 2014) up to 100 epochs and fine-tune (while S is frozen) for up to 50 additional epochs.

For the underlying graph strucure, we consider a grid graph that connects each pixel to itself and its 4 nearest neighbors (or less on the borders). We also consider the square of this graph (pixels are connected to their 13 nearest neighbors, including themselves), the cube of this graph (25 nearest neighbors), up to 10 powers (211 nearest neighbors). We test the model under two setups: either the ordering of the node is unknown, and then we use random one-hot intialization for S; either an ordering of the node is known, and then we use circulant one-hot intialization for S which we freeze in this state. We use the number of nearest neighbors as for the dimension of the first rank of S. We also compare with a convolutional layer of size 5x5, thus containing as many weights as the cube of the grid graph. On MNIST, training architectures that learn S took about twice longer. Table 5 summarizes the obtained results. The ordering is unknown for the first result given, and known for the second result between parenthesis.

Conv5x5	Grid^1	Grid^2	Grid^3
(0.87)	$1.24(1.21)$	$1.02(0.91)$	$0.93(0.91)$
Grid^4	Grid^5	Grid^6	Grid^{10}
$0.90(0.87)$	$0.93(0.80)$	$1.00(0.74)$	$0.93(0.84)$

Table 5: Error rates (in %) on powers of the grid graphs on MNIST.

We observe that even without knowledge of the underlying Euclidean structure, grid GCTs obtain comparable performances as CNNs, and when the ordering is known, they match them. We also noticed that after training, even though the one-hot vectors used for initialization had changed to floating point values, their most significant dimension was almost always the same. That suggests there is room to improve the initialization and the optimization.

In Figure 17, we plot the test error rate for various normalizations when using the square of the grid graph, as a function of the number of epochs of training, only to find that they have little influence on the performance but sometimes improve it a bit. Thus, we will treat them as optional hyperparameters.

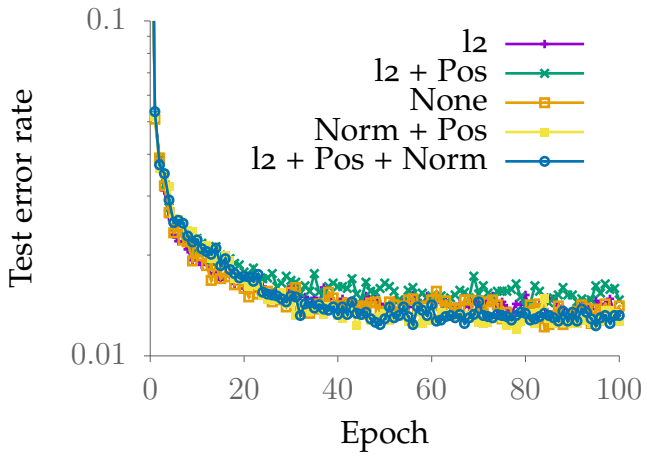


Figure 17: Evolution of the test error rate when learning MNIST using the square of a grid graph and for various normalizations, as a function of the epoch of training. The legend reads: "l2" means ℓ_2 normalization of weights is used (with weights 10^{-5}), "Pos" means parameters in S are forced to being positive, and "Norm" means that the ℓ_1 norm of each vector in the first dimension of S is forced to 1.

3.3.4 Experiments with covariance graphs

As underlying graph structure, we use a thresholded covariance matrix obtained by using all the training examples. We choose the threshold so that the number of remaining edges corresponds to a certain density p (5x5 convolutions correspond approximately to a density of $p = 3\%$). We also infer a graph based on the k nearest neighbors of the inverse of the values of this covariance matrix (k-NN). The latter two are using no prior about the signal underlying structure. The pixels of the input images are

shuffled and the same re-ordering of the pixels is used for every image. Dimension of the first rank of S is chosen equal to k and its weights are initialized random uniformly. GCTs are also compared with models obtained when replacing the first layer by a fully connected or convolutional one. Architecture used is the same as in the previous section. Results are reported on table 6.

MLP	Conv5x5	Thresholded ($p = 3\%$)	k-NN ($k = 25$)
1.44	1.39	1.06	0.96

Table 6: Error rates (in %) on scrambled MNIST.

We observe that GCTs outperform the CNN and the MLP on scrambled MNIST. This is remarkable because that suggests it has been able to exploit information about the underlying structure.

3.3.5 Improved convolutions on shallow architectures

On CIFAR-10 (see Section 1.3.3), we made experiments on shallow CNN architectures and replaced convolutions by receptive graphs. We report results on a variant of AlexNet (Krizhevsky et al., 2012) using little distortion on the input that we forked from a tutorial of tensorflow (Abadi et al., 2015). It is composed of two 5x5 convolutional layers of 64 feature maps, with max pooling and local response normalization (Krizhevsky et al., 2012), followed by two fully connected layers of 384 and 192 neurons. On CIFAR-10, training architectures that learn S took about 2.5 times longer. We compare two different graph supports: the one obtained by using the underlying graph of a regular 5x5 convolution, and the support of the square of the grid graph. Optimization is done with stochastic gradient descent on 375 epochs where S is freezed on the 125 last ones. Circulant

one-hot intialization is used. These are weak classifiers for CIFAR-10 but they are enough to analyse the usefulness of the proposed layer. Experiments are run five times each. Means and standard deviations of accuracies are reported in table 7. "Pos" means parameters in S are forced to being positive, "Norm" means that the ℓ_1 norm of each vector in the third dimension of S is forced to 1, "Both" means both constraints are applied, and "None" means none are used.

Support	Learn S	None	Pos	Norm	Both
Conv5x5	No	/	/	/	86.8 ± 0.2
Conv5x5	Yes	87.4 ± 0.1	87.1 ± 0.2	87.1 ± 0.2	87.2 ± 0.3
Grid^2	Yes	87.3 ± 0.2	87.3 ± 0.1	87.5 ± 0.1	87.4 ± 0.1

Table 7: Accuracies (in %) of shallow networks on CIFAR-10.

The GCTs are able to outperform the corresponding CNNs by a small but statistically significant amount in a shallow architecture.

Learning the scheme S implies a memory overhead due to the increase in the number of weights of each layer, thus why we limited this experiment to a shallow architecture. An example of strategy to extend this experiment to deeper architectures is to tie the schemes of each layer together, or to reuse a same scheme (previously learned) for all layers.

3.3.6 Learning S for semi-supervised node classification

We benchmark multiple graph convolutional methods on citation networks. We refer the reader to Section 1.3.3 for a description of the datasets, and to Section 1.3.5 for a review of the tested models. The models we benchmark are:

- Graph Convolution Network (GCN, Kipf and Welling, 2016),
- Graph Attention Network (GAT, Velickovic et al., 2017),
- Topology Adaptative GCN (TAGCN, Du et al., 2017),
- Addition of graph dropout to GCN (GCN*, from Section 3.2.5),
- Graph Contraction Network (GCT, this work).

To proceed, we forked the official code repository of Velickovic et al., 2017, in order to replicate their environmental setup, from which we implemented the other tested models. This entailed a few differences[1] to be noted between our implementation of GCN and TAGCN from the original ones: a bias is added, the right products XW are computed first and are followed by 50% dropout, and the best model on the validation set is saved to be reused at test time. We also used 50% graph dropout for GAT, TAGCN, and GCT, except on the Pubmed dataset for which we saw that it was detrimental. The inputs of each layer undergo 50% dropout, and the value of graph dropout of GCN* is also 50%. Every model is composed of two hidden layers. GAT models uses 8 heads with 8 hidden units (which amounts to 64 hidden units), whereas the number of hidden units for other models was grid searched in $\{16, 32, 64\}$[2]. The polynomial degree of TAGCN was 2. For comparison, we also ran the experiment with an MLP composed of 500 hidden neurons. The results are reported in Table 8.

[1]Thanks to Daniel Grattarola for his helpful discussion `https://github.com/danielegrattarola/keras-gat/issues/17`.

[2]Most of the time the value of 16 was selected, sometimes 32, and never 64.

Dataset	MLP	GCN	GAT	TAGCN	GCN*	GCT
Cora	58.8 ± 0.9	81.8 ± 0.9	83.3 ± 0.6	82.9 ± 0.7	$\mathbf{83.4} \pm 0.7$	83.3 ± 0.7
Citeseer	56.7 ± 1.1	72.2 ± 0.6	72.1 ± 0.6	71.7 ± 0.7	72.5 ± 0.8	$\mathbf{72.7} \pm 0.5$
Pubmed	72.6 ± 0.9	79.0 ± 0.5	78.3 ± 0.7	78.9 ± 0.5	78.2 ± 0.7	$\mathbf{79.2} \pm 0.4$

Table 8: Mean accuracy (in %) and standard deviation from 100 runs

Even though GCT models performed best overall, the differences with the second bests are not statistically significant. In particular, GCN models enjoy a bump in performances in this setup compared to the experiments in the original paper. The addition of graph dropout put them back on par with the other models on CORA and Citeseer (they were already ahead on Pubmed). We also noticed that the results we obtained for TAGCN on Pubmed were significantly worse than claimed by the authors. We contacted them to check for details and are awaiting response, so we may amend this result in a future version of this document[1]. It is worth noting that the MLP performed a lot worse as expected since it did not exploit the graph structure.

[1]However it is possible that this is due to the experimental setup.

3.4 Inferring the weight sharing scheme

3.4.1 Methodology

As we saw in Section 3.2.4 (Figure 13), the classical convolution on grid graphs can be obtained visually by translating a rectangular window over the pixel domain. The idea of this section is to define similarly a convolution on graph domains, using a notion of translation defined on graphs (Pasdeloup et al., 2017b). The translations we use match Euclidean translations on 2D grid graphs (Grelier et al., 2016), and extend them on general ones, by preserving three simple key properties: injectivity, edge-constraint, and neighborhood-preservation, which we will detail later.
Given a set of graph translations $\mathcal{T} = \{t_i, i \in \{1, 2, \ldots, \kappa - 1\}\} \cup \{t_0 = \mathrm{Id}\}$, the corresponding convolution can be expressed using the same formalism we used in Chapter 2, as:

$$w * x = \sum_{t \in \mathcal{T}} w[t]\, t(x) \tag{65}$$

where x is a signal over vertices of a graph $G = \langle V, E \rangle$, and w is a signal defined on $\mathcal{T} \subset \Phi_{\text{EC}}(G)$. Depending on the algebraic nature of $\mathcal{T}$ (that we will discuss later), the theoretical results obtained in Chapter 2 hold. Under the ternary representation, finding such translations amounts to infer the weight sharing scheme S.
By denoting $w_i = w[t_i]$, we obtain the more familiar expression:

$$\forall u \in V, w * x[u] = \sum_{i=0}^{\kappa-1} w_i\, x[t_i^{-1}(u)] \tag{66}$$

This expression extends the definition of convolutions from grid graphs to general graphs, with the use of graph translations. We use it to obtain extended CNNs that can be applied on graphs. In this manuscript, we

coin the term *Translation-convolutional neural network* (TCNN) to refer to them, but they are the same as the extended convolutions we defined in Pasdeloup et al., 2017a.
Besides defining translation-convolution on general graphs, designing a TCNN also implies extending the other building blocks of classical CNNs. As a counterpart for pooling operations, we define a graph subsampling and apply strided translation-convolution. The translations of the subsampled graph are derived from those of the base graph to define translation-convolution at the downscaled level. We will also use a weak form of data augmentation using these translations. Figure 18 depicts the proposed methodology (Lassance et al., 2018).

3.4.2 Translations

Let a graph $G = \langle V, E \rangle$. We suppose the graph is connected, as conversely the process can be applied to each connected component of G. We denote by d the max degree of the graph and $n = |V|$ the number of vertices.

Definition 98. Candidate-translation
A *candidate-translation* is a function $\phi : U \to V$, where $U \subset V$ and such that:

- ϕ is *injective*:
  $\forall v, v' \in U, \phi(v) = \phi(v') \Rightarrow v = v'$,
- ϕ is *edge-constrained*:
  $\forall v \in U, (v, \phi(v)) \in E$,
- ϕ is *strongly neighborhood-preserving*:
  $\forall v, v' \in U, (v, v') \in E \Leftrightarrow (\phi(v), \phi(v')) \in E$.

The cardinal $|V - U|$ is called the *loss* of ϕ. A translation for which $V = U$ is called a *lossless* translation. Two candidate-translations ϕ and ϕ' are said to be *aligned* if $\exists v \in U, \phi(v) = \phi'(v)$. We define $N_r(v)$ as the set of vertices that are at most r-hop away from a vertex $v \in V$.

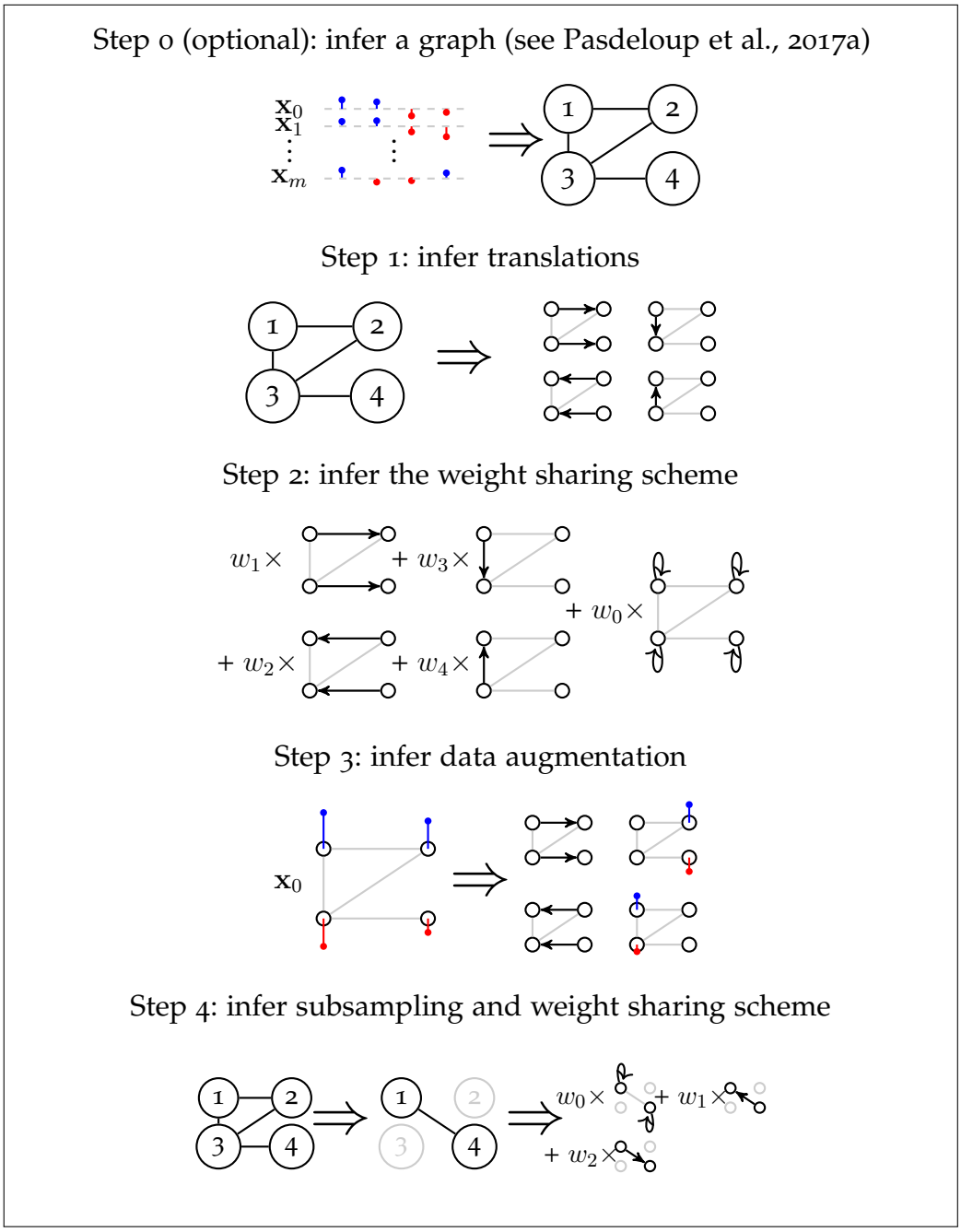


Figure 18: Outline of the proposed method

Definition 99. Translation
A *translation* in a graph G is a candidate-translation such that there is no aligned translation with a strictly smaller loss, or is the identity function.

If the graph is a 2D grid, obtained translations are exactly natural translations on images Grelier et al., 2016.
Because translations and candidate-translations need not be surjective on V, we introduce a zero vertex[1] denoted 0_V, such that any candidate translation $\phi : U \to V$ is extended as a function $\phi : V \to V \cup \{\perp\}$

$$\forall v \notin U, \phi(v) = 0_V \tag{67}$$

Finding translations is an NP-complete problem (Pasdeloup et al., 2017b). So in practice, we will first search locally. For this reason, we define local translations:

Definition 100. Local translation
A *local translation* of center $v \in V$ is a translation in the subgraph of G induced by $N_2(v)$, that has v in its definition domain.

With the help of local translations, we can construct proxies to global translations.

Definition 101. Proxy-translations
A family of *proxy-translations* $(\psi_p)_{p=0,..\kappa-1}$ initialized by $v_0 \in V$ is defined algorithmically as follows:

1. We place an indexing kernel on $N_1(v_0)$ i.e.
   $N_1(v_0) = \{v_0, v_1, ..., v_{\kappa-1}\}$ with $\forall p, \psi_p(v_0) = v_p$,

2. We move this kernel using each local translation ϕ of center v_0:
   $\forall p, \psi_p(\phi(v_0)) = \phi(v_p)$,

[1] Sometimes called *black hole.* (Grelier et al., 2016)

3. We repeat 2) from each new center reached until saturation. If a center is being reached again, we keep the indexing that minimizes the sum of losses of the local translations that has lead to it.

We explain this algorithm in more details in the next Section 3.4.3. A family of proxy-translations defines a translation-convolution as follows:

Definition 102. Translation-convolution layer
Let $(\psi_p)_{p=0,..,\kappa-1}$ be a family of proxy-translations identified on G *s.t.* $\psi_0 = \mathrm{Id}$. The *translation-convolution layer* $\mathcal{L} : x \mapsto y$ is defined as:

$$\forall v \in V, y[v] = h\left(\sum_{p=0}^{\kappa-1} w_p \, x[\psi_p(v)] + b\right)$$

where h is the activation function, b is the bias term, and with the convention that $x[0_V] = 0$.

3.4.3 Finding proxy-translations

We describe in three steps how we efficiently find proxy-translations.

First step: finding local translations
For each vertex $v \in G$, we identify all local translations using a bruteforce algorithm. This process requires finding all translations in all induced subgraphs. There are n such subgraphs, each one contains at most d local translations. Finding a translation can be performed by looking at all possible injections from 1-hop vertices around the central vertex to any vertex that is at most 2-hops away. We conclude that it requires at most $\mathcal{O}(ndd^{2(d+1)})$ elementary operations and is thus linear with the order of the graph. On the other hand, it suggests that sparsity of the graph is a key criterion in order to maintain the complexity reasonable.

Figure 19 depicts an example of a grid graph and the induced subgraph around vertex v_0. Figure 20 depicts all obtained translations in the induced subgraph.

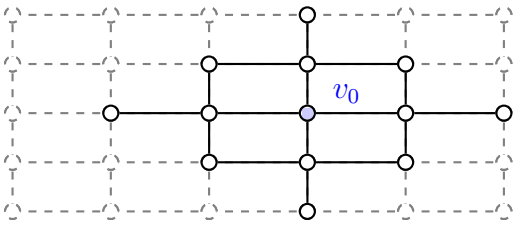


Figure 19: Grid graph (in dashed grey) and the subgraph induced by $N_2(v_0)$ (in black).

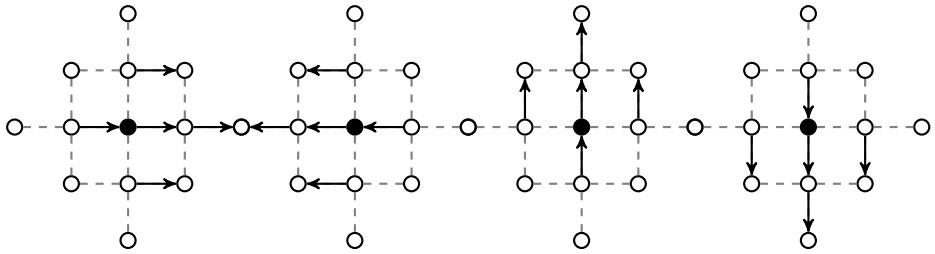


Figure 20: Translations (black arrows) in the induced subgraph (dashed grey) around v_0 (filled in black) that contains v_0 and only some of its neighbors.

Second step: move a small localized kernel

Given an arbitrary[1] vertex $v_0 \in V$, we place an indexing kernel on $N_1(v_0)$ i.e. $N_1(v_0) = \{v_0, v_1, ..., v_{\kappa-1}\}$. Then we move it using every local translations of center v_0, repeating this process for each center that is reached for the first time. We stop when the kernel has been moved everywhere in the graph. In case of multiple paths leading to the same destination,

[1]In practice we run several experiments while changing the initial vertex and keep the best obtained result.

we keep the indexing that minimizes the sum of loss of the series of local translations. We henceforth obtain an indexing of at most κ objects of $N_1(v)$ for every $v \in V$.

This process is depicted in Figure 21. Since it requires moving the kernel everywhere, its complexity is $\mathcal{O}(nd^2)$.

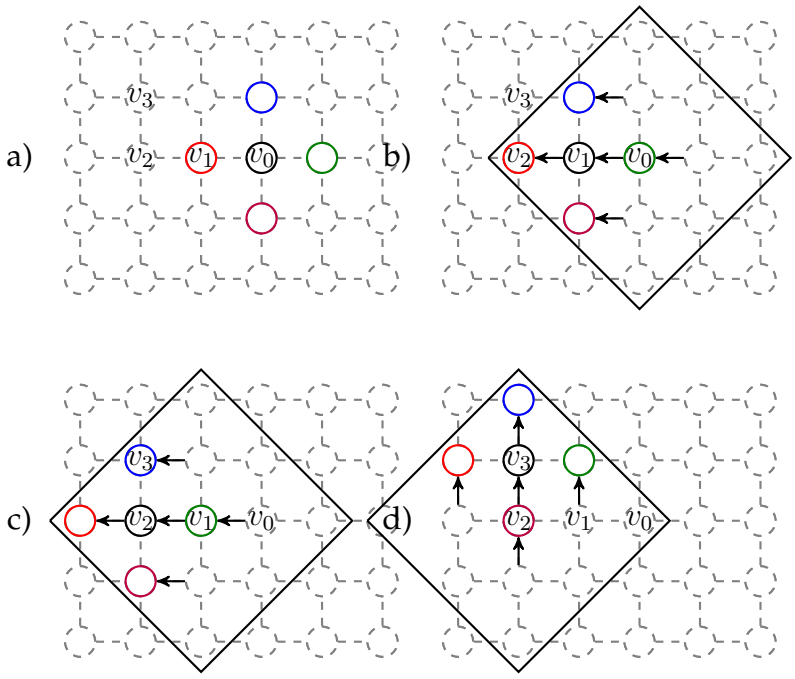


Figure 21: Illustration of the translation of a small indexing kernel using translations in each induced subgraph. Kernel is initialized around v_0 (a), then moved left around v_1 (b) using the induced subgraph around v_0, then moved left again around v_2 (c) using the induced subgraph around v_1 then moved up around v_3 (d) using the induced subgraph around v_2. At the end of the process, the kernel has been localized around each vertex in the graph.

Final step: identifying proxy-translations

Finally, by looking at the indexings obtained in the previous step, we obtain a family of proxy-translations defined globally on G. More precisely,

each index defines its own proxy-translation. Note that they are not translations because only the local properties have been propagated through the second step, so there can exist aligned candidates with smaller losses. Because of the constraint to keep the paths with the minimum sum of losses, they are good proxies to translations on G.
An illustration on a grid graph is given in Figure 22. The complexity is $\mathcal{O}(nd)$. Overall, all three steps are linear in n.

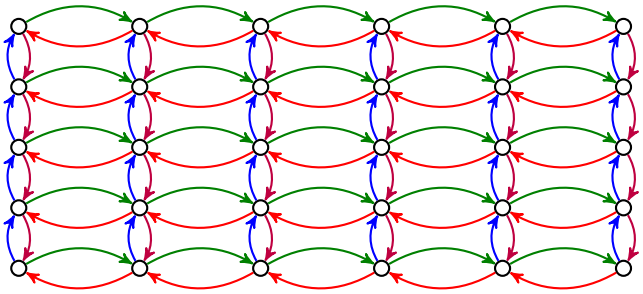


Figure 22: Proxy-translations in G obtained after moving the small kernel around each vertex. Each color corresponds to one translation.

3.4.4 Subsampling

Downscaling is a tricky part of the process because it supposes one can somehow regularly sample vectors. As a matter of fact, a nonregular sampling is likely to produce a highly irregular downscaled graph, on which looking for translations irremediably leads to poor accuracy, as we noticed in our experiments.
We rather define the translations of the strided graph using the previously found proxy-translations on G.

First step: extended convolution with stride r
Given an arbitrary initial vertex $v_0 \in V$, the set of kept vertices $V_{\downarrow r}$ is defined inductively as follows:

- $V_{\downarrow r}^0 = \{v_0\}$,
- $\forall t \in \mathbb{N}, V_{\downarrow r}^{t+1} = V_{\downarrow r}^t \cup \{v \in V, \forall v' \in V_{\downarrow}^t, v \notin N_{r-1}(v') \wedge \exists v' \in V_{\downarrow r}^t, v \in N_r(v')\}$.

This sequence is nondecreasing and bounded by V, so it eventually becomes stationary and we obtain $V_{\downarrow r} = \lim_t V_{\downarrow r}^t$. $V_{\downarrow r}$ is the set of output neurons of the extended convolution layer with stride r. Figure 23 illustrate the first downscaling $V_{\downarrow 2}$ on a grid graph.

Second step: convolutions for the strided graph

Using the proxy-translations on G, we move a localized r-hop indexing kernel over G. At each location, we associate the vertices of $V_{\downarrow r}$ with indices of the kernel, thus obtaining what we define as induced $_{\downarrow r}$-translations on the set $V_{\downarrow r}$. In other words, when the kernel is centered on v_0, if $v_1 \in V_{\downarrow r}$ is associated with the index p_0, we obtain $\phi_{p_0}^{\downarrow r}(v_0) = v_1$. Subsequent convolutions at lower scales are defined using these induced $_{\downarrow r}$-translations.

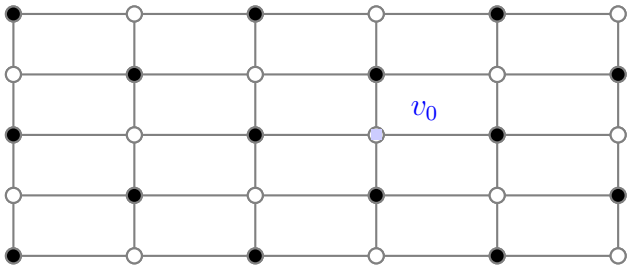


Figure 23: Downscaling of the grid graph. Disregarded vertices are filled.

3.4.5 Data augmentation

Once proxy-translations are obtained on G, we use them to move training signals, artificially creating new ones. Note that this type of data-augmentation is weaker than for images since no flipping, scaling or rotations are used.

3.4.6 Experiments

Matching CNNs on an image classification task

On the CIFAR-10 dataset (see Section 1.3.3), our models are based on a variant of a deep residual network, namely PreActResNet18 (He et al., 2016b). We tested different combinations of graph support and data augmentation. For the graph support, we use either a regular 2D grid or either an inferred graph obtained by keeping the four neighbours that covary the most. In the second case, which correspond to scrambled CIFAR-10 (see Section 1.3.3), no structure prior have been fed to the process, so the results can be compared with those of Lin et al., 2015. We also report results obtained with ChebNet (Defferrard et al., 2016), where only convolutional layers have been replaced for comparison.

Table 9 summarizes our results. In particular, it is interesting to note that results obtained without any structure prior (91.07%) are only 2.7% away from the baseline using classical CNNs on images (93.80%). This gap is even smaller (less than 1%) when using the grid prior. Also, without priors our method significantly outperforms the others.

Experiments on a fMRI dataset

We used a shallow network on the PINES dataset (see Section 1.3.3). The results reported on Table 10 show that our method was able to improve over CNNs, MLPs and other graph-based extended convolutional neural networks.

Support	MLP	CNN	Grid Graph		Covariance Graph
			ChebNet[c]	Proposed	Proposed
Full Data Augmentation	78.62%[a,b]	**93.80%**	85.13%	93.94%	92.57%
Data Augmentation - Flip	——	92.73%	84.41%	92.94%	91.29%
Graph Data Augmentation	——	92.10%[d]	——	92.81%	**91.07%**[a]
None	69.62%	87.78%	——	88.83%	85.88%[a]

[a] No priors about the structure
[b] Lin et al., 2015
[c] Defferrard et al., 2016
[d] Data augmentation done with covariance graph

Table 9: CIFAR-10 result comparison table.

Support	None		Neighborhood Graph	
Method	MLP	CNN (1x1 kernels)	ChebNet[c]	Proposed
Accuracy	82.62%	84.30%	82.80%	**85.08%**

Table 10: PINES fMRI dataset accuracy comparison table.

Conclusion

In this manuscript, after presenting the domains related to our subject, we built a theory for convolutions on graphs, in view of using them in CNN on graph domains. On enclidean domains, convolutional layers take advantage of the translational equivariances of the convolution. Therefore, our construction on graph domains depends on a set of transformations of the vertex set for which the resulting operation is also equivariant. More precisely, we wanted to define a class of convolutional operators that are exactly the class of linear operators that are equivariant to this set of transformation. We demonstrated that this characterization holds should this set have an algebraic structure of group, groupoid or path groupoid. In particular, we proved that this amounts to search for Cayley subgraphs. We also saw that the possible abelianity of these structures is linked with the property that the convolution is supported locally. Then, we studied neural networks intended for graph domains. We adopted an approach based on graph representations of the propagation between layers of neurons. We proved that if the local receptive fields of the neurons are intertwined, then if their input have a graph structure, it can be used to define the propagation. We also discovered that the linear part of a layer can be expressed by an operator that involves three operands: the input signal, the weight kernel and the weight sharing scheme. We called it *neural contraction*, in reference to the term *tensor contraction*. We showed that it is associative, commutative, and generic in the sense that it can represent any kind of layer. We used this representation to see the influence

of symmetries that are present in the structure of the data. We conducted experiments to learn how the weights are shared in addtion of learning the weights. In a sense, this amounts, in the case of convolutions, to try to learn its set of transformations to which it is equivariant. We saw that it attains similar performances than other state-of-the-art models. Then we made experiments for a CNN for which the convolution is based on a set of translations on graphs, which defines how the weights are shared. We proposed an alogrithm to find the translations. We defined downscaling and data augmentation from these translations, and used a residual network architecture. We showed that this model retrieves the performances of CNNs without feeding to it the underlying structure of images, and that it attains strong performances on graph signal datasets.

In conclusion, we proposed a novel layer representation for extending CNN architectures to other input domains than those for what they where intended. This, as pointed out in the introduction, participates in rending them more generic, and thus applicable to a broader range of real world problems. In the process, we also advanced our understanding of convolutions. We hope that the reader had pleasure reading this manuscript and that it gave him ideas and shed new lights. Let us thrive for a continuous effort to help advance collectively the boundaries of human knowledge at our scales and beyond !

Bibliography

Abadi, Martín, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: `http://tensorflow.org/` (cit. on pp. 7, 29, 126).

Arora, Raman, Amitabh Basu, Poorya Mianjy, and Anirbit Mukherjee (2018). "Understanding Deep Neural Networks with Rectified Linear Units". In: *International Conference on Learning Representations*. URL: `https://openreview.net/forum?id=B1J_rgWRW` (cit. on p. 26).

Atwood, James and Don Towsley (2016). "Diffusion-convolutional neural networks". In: *Advances in Neural Information Processing Systems*, pp. 1993–2001 (cit. on p. 45).

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (cit. on p. 46).

Bai, Lu, Yuhang Jiao, Luca Rossi, Lixin Cui, Jian Cheng, and Edwin R Hancock (2018). "A Quantum Spatial Graph Convolutional Neural Network using Quantum Passing Information". In: *arXiv preprint arXiv:1809.01090* (cit. on p. 37).

Bass, Jean (1968). "Cours de mathématiques". In: (cit. on p. 5).

Bengio, Yoshua (2009). "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on p. 24).

Bianchini, Monica and Franco Scarselli (2014). "On the complexity of neural network classifiers: A comparison between shallow and deep architectures". In: *IEEE transactions on neural networks and learning systems* 25.8, pp. 1553–1565 (cit. on p. 26).

Brandt, Heinrich (1927). "Über eine Verallgemeinerung des Gruppenbegriffes". In: *Mathematische Annalen* 96.1, pp. 360–366 (cit. on p. 82).

Bronstein, Michael M, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst (2017). "Geometric deep learning: going beyond euclidean data". In: *IEEE Signal Processing Magazine* 34.4, pp. 18–42 (cit. on p. 33).

Bruna, Joan and Stéphane Mallat (2013). "Invariant scattering convolution networks". In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1872–1886 (cit. on p. 44).

Bruna, Joan, Wojciech Zaremba, Arthur Szlam, and Yann LeCun (2013). "Spectral networks and locally connected networks on graphs". In: *arXiv preprint arXiv:1312.6203* (cit. on p. 41).

Cayley, Professor (1878). "Desiderata and Suggestions: No. 2. The Theory of Groups: Graphical Representation". In: *American Journal of Mathematics* 1.2, pp. 174–176. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2369306 (cit. on p. 74).

Chang, Luke J, Peter J Gianaros, Stephen B Manuck, Anjali Krishnan, and Tor D Wager (2015). "A sensitive and specific neural signature for

picture-induced negative affect". In: *PLoS biology* 13.6, e1002180 (cit. on p. 38).

Chellapilla, Kumar, Sidd Puri, and Patrice Simard (2006). "High performance convolutional neural networks for document processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft (cit. on p. 113).

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang (2015). "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv preprint arXiv:1512.01274* (cit. on p. 7).

Chen, Xu, Xiuyuan Cheng, and Stéphane Mallat (2014). "Unsupervised deep haar scattering on graphs". In: *Advances in Neural Information Processing Systems*, pp. 1709–1717 (cit. on p. 44).

Chetlur, Sharan, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer (2014). "cudnn: Efficient primitives for deep learning". In: *arXiv preprint arXiv:1410.0759* (cit. on p. 114).

Chollet, François et al. (2015). *Keras*. `https://keras.io` (cit. on p. 7).

Chollet, François (2016). "Xception: Deep Learning with Depthwise Separable Convolutions". In: *arXiv preprint arXiv:1610.02357* (cit. on p. 118).

Chung, Fan R. K. (1996). *Spectral Graph Theory (CBMS Regional Conference Series in Mathematics, No. 92)*. American Mathematical Society. ISBN: 0821803158 (cit. on p. 40).

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2015). "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (cit. on p. 25).

Cohen, Nadav and Amnon Shashua (2016). "Convolutional rectifier networks as generalized tensor decompositions". In: *International Conference on Machine Learning*, pp. 955–963 (cit. on p. 26).

Cohen, Nadav, Ronen Tamari, and Amnon Shashua (2018). "Boosting Dilated Convolutional Networks with Mixed Tensor Decompositions". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=S1JHhv6TW (cit. on p. 26).

Cord, Matthieu (2016). *Deep learning an weak supervision for image classification*. [Online; accessed April-2018]. URL: http://webia.lip6.fr/~cord/pdfs/news/TalkDeepCordI3S.pdf (cit. on p. 17).

Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4, pp. 303–314 (cit. on pp. 24, 25).

D'Alembert, Jean Le Rond (1754). *Recherche sur différents points importants du système du monde* (cit. on p. 1).

Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst (2016). "Convolutional neural networks on graphs with fast localized spectral filtering". In: *Advances in Neural Information Processing Systems*, pp. 3837–3845 (cit. on pp. 39, 42–44, 110, 119, 139, 140).

Delalleau, Olivier and Yoshua Bengio (2011). "Shallow vs. deep sum-product networks". In: *Advances in Neural Information Processing Systems*, pp. 666–674 (cit. on p. 26).

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei (2009). "Imagenet: A large-scale hierarchical image database". In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, pp. 248–255 (cit. on pp. 20, 24).

Dominguez-Torres, Alejandro (2010). "The origin and history of convolution: continuous and discrete convolution operations". In: [Online; accessed April-2018]. URL: http://www.slideshare.net/Alexdfar/origin-adn-history-of-convolution (cit. on p. 1).

Du, Jian, Shanghang Zhang, Guanhang Wu, José MF Moura, and Soummya Kar (2017). "Topology adaptive graph convolutional networks". In: *arXiv preprint arXiv:1710.10370* (cit. on pp. 47, 110, 128).

Eldan, Ronen and Ohad Shamir (2016). "The power of depth for feedforward neural networks". In: *Conference on Learning Theory*, pp. 907–940 (cit. on p. 26).

Exel, Ruy (1998). "Partial actions of groups and actions of inverse semigroups". In: *Proceedings of the American Mathematical Society* 126.12, pp. 3481–3494 (cit. on p. 91).

Gama, Fernando, Alejandro Ribeiro, and Joan Bruna (2018). "Diffusion Scattering Transforms on Graphs". In: *arXiv preprint arXiv:1806.08829* (cit. on p. 44).

Gilmer, Justin, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl (2017). "Neural message passing for quantum chemistry". In: *arXiv preprint arXiv:1704.01212* (cit. on p. 45).

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256 (cit. on pp. 24, 123).

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio (2011). "Deep sparse rectifier neural networks". In: *International Conference on Artificial Intelligence and Statistics*, pp. 315–323 (cit. on pp. 16, 25).

Grelier, Nicolas, Bastien Pasdeloup, Jean-Charles Vialatte, and Vincent Gripon (2016). "Neighborhood-Preserving Translations on Graphs". In: *Proceedings of IEEE GlobalSIP*, pp. 410–414 (cit. on pp. 130, 133).

Grover, Aditya and Jure Leskovec (2016). "node2vec: Scalable feature learning for networks". In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 855–864 (cit. on p. 37).

Hackbusch, Wolfgang (2012). *Tensor spaces and numerical tensor calculus*. Vol. 42. Springer Science & Business Media (cit. on pp. 5, 6).

Hamilton, Will, Zhitao Ying, and Jure Leskovec (2017a). "Inductive representation learning on large graphs". In: *Advances in Neural Information Processing Systems*, pp. 1024–1034 (cit. on p. 37).

Hamilton, William L, Rex Ying, and Jure Leskovec (2017b). "Representation learning on graphs: Methods and applications". In: *arXiv preprint arXiv:1709.05584* (cit. on p. 37).

Hammond, David K, Pierre Vandergheynst, and Rémi Gribonval (2011). "Wavelets on graphs via spectral graph theory". In: *Applied and Computational Harmonic Analysis* 30.2, pp. 129–150 (cit. on pp. 41, 43, 44).

Håstad, Johan and Mikael Goldmann (1991). "On the power of small-depth threshold circuits". In: *Computational Complexity* 1.2, pp. 113–129 (cit. on p. 25).

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034 (cit. on p. 25).

— (2016a). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778 (cit. on pp. 14, 19, 20).

— (2016b). "Identity mappings in deep residual networks". In: *European Conference on Computer Vision*. Springer, pp. 630–645 (cit. on p. 139).

Hechtlinger, Yotam, Purvasha Chakravarti, and Jining Qin (2017). "A generalization of convolutional neural networks to graph-structured data". In: *arXiv preprint arXiv:1704.08165* (cit. on p. 45).

Henaff, Mikael, Joan Bruna, and Yann LeCun (2015). "Deep convolutional networks on graph-structured data". In: *arXiv preprint arXiv:1506.05163* (cit. on p. 44).

Hinton, Geoffrey, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. (2012). "Deep neural networks for

acoustic modeling in speech recognition: The shared views of four research groups". In: *IEEE Signal Processing Magazine* 29.6, pp. 82–97 (cit. on p. 14).

Hinton, Geoffrey E, Simon Osindero, and Yee-Whye Teh (2006). "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7, pp. 1527–1554 (cit. on p. 24).

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2, pp. 251–257 (cit. on p. 24).

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366 (cit. on pp. 24, 27).

Huang, Gao, Zhuang Liu, Kilian Q Weinberger, and Laurens van der Maaten (2017). "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. Vol. 1. 2, p. 3 (cit. on pp. 14, 19, 20).

Ioffe, Sergey and Christian Szegedy (2015). "Batch normalization: accelerating deep network training by reducing internal covariate shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37*. JMLR. org, pp. 448–456 (cit. on p. 25).

Jaderberg, Max, Karen Simonyan, Andrew Zisserman, et al. (2015). "Spatial transformer networks". In: *Advances in neural information processing systems*, pp. 2017–2025 (cit. on p. 43).

Jarrett, Kevin, Koray Kavukcuoglu, Yann LeCun, et al. (2009). "What is the best multi-stage architecture for object recognition?" In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, pp. 2146–2153 (cit. on p. 25).

Joachims, Thorsten (1996). *A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization.* Tech. rep. Carnegie-mellon univ pittsburgh pa dept of computer science (cit. on pp. 39, 43).

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (cit. on p. 123).

Kipf, Thomas N and Max Welling (2016). "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (cit. on pp. 39, 45, 110, 120, 128).

Klambauer, Günter, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter (2017). "Self-Normalizing Neural Networks". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Curran Associates, Inc., pp. 971–980. URL: http://papers.nips.cc/paper/6698-self-normalizing-neural-networks.pdf (cit. on p. 25).

Krizhevsky, Alex (2009). "Learning multiple layers of features from tiny images". In: (cit. on p. 38).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*, pp. 1097–1105 (cit. on pp. 14, 25, 28, 126).

Lassance, Carlos Eduardo Rosar Kos, Jean-Charles Vialatte, and Vincent Gripon (2018). "Matching Convolutional Neural Networks without Priors about Data". In: (cit. on pp. 38, 131).

LeCun, Y. (1987). "Modeles connexionnistes de l'apprentissage (connectionist learning models)". PhD thesis. Université P. et M. Curie (Paris 6) (cit. on p. 20).

LeCun, Yann, Yoshua Bengio, et al. (1995). "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10, p. 1995 (cit. on pp. 14, 28).

LeCun, Yann, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel (1989). "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4, pp. 541–551 (cit. on pp. 14, 16, 17, 25).

LeCun, Yann, Corinna Cortes, and Christopher JC Burges (1998). *The MNIST database of handwritten digits* (cit. on pp. 38, 117).

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton (2015). "Deep learning". In: *Nature* 521.7553, pp. 436–444 (cit. on p. 1).

Lee, John Boaz, Ryan A Rossi, Sungchul Kim, Nesreen K Ahmed, and Eunyee Koh (2018). "Attention Models in Graphs: A Survey". In: *arXiv preprint arXiv:1807.07984* (cit. on pp. 45, 46).

Levie, Ron, Federico Monti, Xavier Bresson, and Michael M Bronstein (2017). "CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters". In: *arXiv preprint arXiv:1705.07664* (cit. on p. 44).

Li, Xin (2016). "Partial transformation groupoids attached to graphs and semigroups". In: *International Mathematics Research Notices* 2017.17, pp. 5233–5259 (cit. on p. 91).

Lin, Henry W, Max Tegmark, and David Rolnick (2017). "Why does deep and cheap learning work so well?" In: *Journal of Statistical Physics* 168.6, pp. 1223–1247 (cit. on p. 26).

Lin, Zhouhan, Roland Memisevic, and Kishore Reddy Konda (2015). "How far can we go without convolution: Improving fully-connected networks". In: *CoRR* abs/1511.02580. arXiv: 1511.02580. URL: http://arxiv.org/abs/1511.02580 (cit. on pp. 57, 139, 140).

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng (2013). "Rectifier nonlinearities improve neural network acoustic models". In: *Proceedings of the 30th international conference on machine learning* (cit. on p. 25).

Marcus, Marvin (1975). "Finite dimensional multilinear algebra". In: (cit. on p. 5).

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133 (cit. on p. 21).

Mhaskar, Hrushikesh, Qianli Liao, and Tomaso Poggio (2016). "Learning functions: when is deep better than shallow". In: *arXiv preprint arXiv:1603.00988* (cit. on p. 26).

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean (2013a). "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*, pp. 3111–3119 (cit. on p. 37).

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean (2013b). "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (cit. on p. 37).

Monti, Federico, Davide Boscaini, Jonathan Masci, Emanuele Rodolà, Jan Svoboda, and Michael M Bronstein (2016). "Geometric deep learning on graphs and manifolds using mixture model CNNs". In: *arXiv preprint arXiv:1611.08402* (cit. on pp. 45, 110).

Montufar, Guido F, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio (2014). "On the number of linear regions of deep neural networks". In: *Advances in neural information processing systems*, pp. 2924–2932 (cit. on p. 26).

Ng, Andrew Y (2004). "Feature selection, L 1 vs. L 2 regularization, and rotational invariance". In: *Proceedings of the twenty-first international conference on Machine learning*. ACM, p. 78 (cit. on p. 123).

Nica, Alexandru (1994). "On a groupoid construction for actions of certain inverse semigroups". In: *International Journal of Mathematics* 5.03, pp. 349–372 (cit. on p. 91).

Nickolls, John, Ian Buck, Michael Garland, and Kevin Skadron (2008). "Scalable parallel programming with CUDA". In: *ACM SIGGRAPH 2008 classes*. ACM, p. 16 (cit. on p. 25).

Niepert, Mathias and Alberto Garcia-Duran (2018). "Towards a Spectrum of Graph Convolutional Networks". In: *arXiv preprint arXiv:1805.01837* (cit. on p. 47).

Niepert, Mathias, Mohamed Ahmed, and Konstantin Kutzkov (2016). "Learning Convolutional Neural Networks for Graphs". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pp. 2014–2023 (cit. on pp. 37, 45).

Nikolentzos, Giannis, Polykarpos Meladianos, Antoine Jean-Pierre Tixier, Konstantinos Skianis, and Michalis Vazirgiannis (2017). "Kernel Graph Convolutional Neural Networks". In: *arXiv preprint arXiv:1710.10689* (cit. on p. 37).

Oliphant, Travis E (2006). *A guide to NumPy*. Vol. 1. Trelgol Publishing USA (cit. on p. 7).

Orhan, Emin and Xaq Pitkow (2018). "Skip Connections Eliminate Singularities". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=HkwBEMWCZ (cit. on p. 27).

Pan, Xingyuan and Vivek Srikumar (2016). "Expressiveness of rectifier networks". In: *International Conference on Machine Learning*, pp. 2427–2435 (cit. on p. 26).

Pascanu, Razvan, Guido Montufar, and Yoshua Bengio (2013). "On the number of response regions of deep feed forward networks with piecewise linear activations". In: *arXiv preprint arXiv:1312.6098* (cit. on pp. 25, 26).

Pasdeloup, Bastien, Vincent Gripon, Jean-Charles Vialatte, and Dominique Pastor (2017a). "Convolutional neural networks on irregular domains through approximate translations on inferred graphs". In: *arXiv preprint arXiv:1710.10035* (cit. on pp. 45, 131, 132).

Pasdeloup, Bastien, Vincent Gripon, Nicolas Grelier, Jean-Charles Vialatte, and Dominique Pastor (2017b). "Translations on graphs with neighborhood preservation". In: *arXiv preprint arXiv:1709.03859* (cit. on pp. 130, 133).

Paszke, Adam, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga,

and Adam Lerer (2017). "Automatic differentiation in PyTorch". In: (cit. on p. 7).

Poggio, Tomaso, Fabio Anselmi, and Lorenzo Rosasco (2015). *I-theory on depth vs width: hierarchical function composition*. Tech. rep. Center for Brains, Minds and Machines (CBMM) (cit. on p. 26).

Poole, Ben, Subhaneil Lahiri, Maithra Raghu, Jascha Sohl-Dickstein, and Surya Ganguli (2016). "Exponential expressivity in deep neural networks through transient chaos". In: *Advances in neural information processing systems*, pp. 3360–3368 (cit. on p. 26).

Raghu, Maithra, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein (2016). "On the expressive power of deep neural networks". In: *arXiv preprint arXiv:1606.05336* (cit. on p. 26).

Robbins, Herbert and Sutton Monro (1985). "A stochastic approximation method". In: *Herbert Robbins Selected Papers*. Springer, pp. 102–109 (cit. on p. 22).

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1985). *Learning internal representations by error propagation*. Tech. rep. California Univ San Diego La Jolla Inst for Cognitive Science (cit. on pp. 14, 21).

Sandryhaila, Aliaksei and José MF Moura (2013). "Discrete signal processing on graphs". In: *IEEE transactions on signal processing* 61.7, pp. 1644–1656 (cit. on p. 47).

Sankar, Aravind, Xinyang Zhang, and Kevin Chen-Chuan Chang (2017). "Motif-based Convolutional Neural Network on Graphs". In: *arXiv preprint arXiv:1711.05697* (cit. on p. 45).

Schwartz, Laurent (1957). *Théorie des distributions*. Vol. 2. Hermann Paris (cit. on p. 54).

Sen, Prithviraj, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad (2008). "Collective classification in network data". In: *AI magazine* 29.3, p. 93 (cit. on p. 39).

Shuman, David I, Sunil K Narang, Pascal Frossard, Antonio Ortega, and Pierre Vandergheynst (2013). "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, pp. 83–98 (cit. on p. 40).

Simonovsky, Martin and Nikos Komodakis (2017). "Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs". In: *arXiv preprint arXiv:1704.02901* (cit. on p. 45).

Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (cit. on pp. 14, 17, 18).

Srivastava, Nitish, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting." In: *Journal of Machine Learning Research* 15.1, pp. 1929–1958 (cit. on p. 25).

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. (2015). "Going deeper with convolutions". In: *Conference on Computer Vision and Pattern Recognition* (cit. on pp. 14, 20).

Tixier, Antoine Jean-Pierre, Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis (2017). "Classifying Graphs as Images with Convolutional Neural Networks". In: *arXiv preprint arXiv:1708.02218* (cit. on p. 37).

Van Den Oord, Aaron, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu (2016). "Wavenet: A generative model for raw audio". In: *arXiv preprint arXiv:1609.03499* (cit. on p. 27).

Velickovic, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio (2017). "Graph Attention Networks". In: *stat* 1050, p. 20 (cit. on pp. 45, 46, 110, 128).

Vialatte, Jean-Charles, Vincent Gripon, and Grégoire Mercier (2016). "Generalizing the convolution operator to extend cnns to irregular domains". In: *arXiv preprint arXiv:1606.01166* (cit. on pp. 45, 110, 116).

Vialatte, Jean-Charles, Vincent Gripon, and Gilles Coppin (2017). "Learning Local Receptive Fields and their Weight Sharing Scheme on Graphs". In: *arXiv preprint arXiv:1706.02684* (cit. on pp. 45, 121, 122).

Weinstein, Alan (1996). "Groupoids: unifying internal and external symmetry". In: *Notices of the AMS* 43.7, pp. 744–752 (cit. on p. 82).

Widrow, Bernard and Marcian E Hoff (1960). *Adaptive switching circuits*. Tech. rep. STANFORD UNIV CA STANFORD ELECTRONICS LABS (cit. on p. 21).

Wikipedia, contributors (2018a). *Feedforward neural network — Wikipedia, The Free Encyclopedia*. [Online; accessed April-2018]. URL: https://en.wikipedia.org/wiki/Feedforward_neural_network (cit. on p. 14).

— (2018b). *Softmax function — Wikipedia, The Free Encyclopedia*. [Online; accessed April-2018]. URL: https://en.wikipedia.org/wiki/Softmax_function (cit. on p. 16).

Williamson, S Gill (2015). "Tensor spaces-the basics". In: *arXiv preprint arXiv:1510.02428* (cit. on p. 5).

Yamada, Yoshihiro, Masakazu Iwamura, and Koichi Kise (2018). "ShakeDrop regularization". In: *arXiv preprint arXiv:1802.02375* (cit. on p. 57).

Yang, Zhilin, William W Cohen, and Ruslan Salakhutdinov (2016). "Revisiting semi-supervised learning with graph embeddings". In: *arXiv preprint arXiv:1603.08861* (cit. on p. 39).

Yi, Li, Hao Su, Xingwen Guo, and Leonidas Guibas (2016). "Syncspeccnn: Synchronized spectral CNN for 3d shape segmentation". In: *arXiv preprint arXiv:1612.00606* (cit. on p. 43).

Zell, Andreas (1994). *Simulation neuronaler netze*. Vol. 1. Addison-Wesley Bonn (cit. on p. 14).

Zermelo, Ernst (1904). "Beweis, daß jede Menge wohlgeordnet werden kann". In: *Mathematische Annalen* 59.4, pp. 514–516 (cit. on p. 89).

Zhang, Daokun, Jie Yin, Xingquan Zhu, and Chengqi Zhang (2017). "Network representation learning: A survey". In: *IEEE transactions on Big Data* (cit. on p. 37).

Zoph, Barret and Quoc V Le (2016). "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (cit. on p. 14).

Zou, D. and G. Lerman (2018). "Graph Convolutional Neural Networks via Scattering". In: *ArXiv e-prints*. arXiv: 1804.00099 [cs.IT] (cit. on p. 44).