Contents

Introduction

TODO:

Chapter 1

Presentation of the field

Contents

Introduction

In this chapter, we present notions related to our domains of interest. In particular, for tensors we give original definitions that are more appropriate for our study. In the neural network's section, we present the concepts necessary to understand the evolution of the state of the art research in this field. In the last section, we present graphs for their usage in deep learning.
Vector spaces considered in what follows are assumed to be finite-dimensional and over the field of real numbers $\mathbb{R}$.

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 image of a basis can be represented by a multidimensional array. Alternatively, Hackbush propose 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 depend 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, for that they are embedded in a vector 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 other 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.

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 canon 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
For naming convenience, from now on, we will distinguish between the terms *linear space* and *vector space i.e.* we will abusively use the term *vector space* only to refer to a linear space that is seen as a tensor space of rank 1. If we don't know its rank, we rather use the term *linear space*. 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* instead of *rank* (*e.g.* Hackbusch, 2012) as the latter is 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.

Definition 4. Indexing

An *entry* of a tensor $t \in \mathbb{T}$ is one of its scalar coordinates in the canonical basis, denoted $t[i_1, i_2, \dots, i_r]$.

More precisely, if $\mathbb{T} = \bigotimes_{k=1}^{r} \mathbb{V}_k$, with bases $((e_k^i)_{i=1,\dots,n_k})_{k=1,\dots,r}$, then we have

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

The cartesian product $\mathbb{I} = \prod_{k=1}^{r} [\![1, n_k]\!]$ is called the *index space* of $\mathbb{T}$

Remark. When using an index i_k for an entry of a tensor t, we implicitly assume that $i_k \in [\![1, n_k^{(t)}]\!]$ if nothing is specified.

Definition 5. Subtensor

A *subtensor* t' is a tensor of same rank composed of entries of t that are contiguous in the indexing, with at least one entry per rank. We denote $t' = t[l_1{:}u_1, l_2{:}u_2, \dots, l_r{:}u_r]$, where the $\{l_k\}$ and the $\{u_k\}$ are the lower and upper bounds of the indices used by the entries that compose t'.

Remark. We don't necessarily write the lower bound index if it is equal to 1, neither the upper bound index if it is equal to $n_k^{(t)}$.

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:

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

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 (see Section 1.2.7).

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:

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

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

Using subscript notation for covariant indices and superscript notation for contravariant indices, the previous tensor contraction can be written using the Einstein summation convention as:

$$t_{1_{i_1^{(1)}\cdots i_{r_1-s}^{(1)}}}{}^{k_1\cdots k_s} t_{2_{k_1\cdots k_s}}{}^{i_{s+1}^{(2)}\cdots i_{r_2}^{(2)}} = t_{3_{i_1^{(1)}\cdots i_{r_1-s}^{(1)}}}{}^{i_{s+1}^{(2)}\cdots i_{r_2}^{(2)}} \tag{2}$$

Dot product $u_k v^k = \lambda$ and matrix product $A_i{}^k B_k{}^j = C_i{}^j$ are common examples of tensor contractions.

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:

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

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 $t'_1[i_1, \dots, i_n, k_1, \dots, k_n] = t_1[i_1 + n_1^{(2)} - k_1, \dots, i_n + n_n^{(2)} - k_n]$, then

$$t_3[i_1, \dots, i_n] = \sum_{k_1=1}^{n_1^{(2)}} \cdots \sum_{k_n=1}^{n_n^{(2)}} t'_1[i_1, \dots, i_n, k_1, \dots, k_n]\, t_2[k_1, \dots, k_n]$$

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 $s = (s_1, s_2, \dots, s_n)$, 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:

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

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

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 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 and Bengio, 1995), even up until the important breakthroughs that generated a surge of interest in the field (Hinton et al., 2012; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). However, in more recent advances, more complex architectures have emerged (Szegedy et al., 2015; He et al., 2016; 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, 2018c).

1.2.1 Simple formalization

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 $I_f \xrightarrow{f} O_f$.

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 on any linear space with the functional notation $h(v)[i] = h(v[i])$.

Definition 18. Layer
A couple (g, h), where g is an affine or linear function, and h is an activation function is called a *layer*. The set of layers is denoted $\mathcal{L}$.

Adoption of ReLU activations
Historically, sigmoidal and tanh activations were mostly used (Cybenko, 1989; LeCun et al., 1989). However in recent practice, the *rectified linear unit* (ReLU), which implements the *rectifier* function $h : x \mapsto max(0, x)$ with convention $h'(0) = 0$ (first introduced as the *positive part*, Jarrett et al., 2009), is 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).

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, for the application of supervised learning when a neural network is trained from data (see Section 1.2.4), this result is quite important because

it brings theoretical justification that the objective exists (even though it doesn't 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 a deep convolutional architecture (Krizhevsky et al., 2012), see Section 1.2.7. 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.

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., 2018a).

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 net-

work) 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).

Bias

Note that affine functions $\widetilde{g}$ can be written as a sum between a linear function g and a constant vector b which is called the *bias*. It augments the expressivity of the neural network's family of functions. For notational convenience, we will often omit to write down the biases in the layer's equations.

1.2.2 Generic 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'll 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.

A *branching operation* between n tensors, $x_{k_1} \bowtie x_{k_2} \bowtie \cdots \bowtie x_{k_n}$, is a composition of binary branching operations, or is the identity function Id if $n = 1$. Branching operations are also naturally defined on tensor-valued functions through their realizations.

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

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., 2016) 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. 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.

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., 2018a) have analyzed the impact of residual connections used in Wavenet-like architectures (Van Den Oord et al., 2016) in terms of expressive efficiency (see related paragraph

in Section 1.2.1) using tools from the field of tensor analysis ; (Orhan and Pitkow, 2018) have empirically demonstrated that skip connections can resolve some inefficiency problems inherent of fully-connected networks (dead activations, activations that are always equal, linearly dependent sets of activations).

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.

1.2.3 Interpretation

Until now, we have formally introduced a neural network as a mathematical function. As its name suggests, such function can be interpreted from a connectivity viewpoint (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 *(computational) neuron* is 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 a derivable activation,
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 1.1.

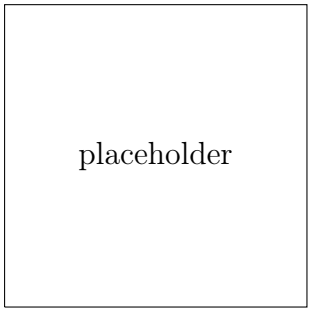

Figure 1.1: A neuron

1.2.4 Parameterization and training

TODO: pass on this section

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 in the opposite direction of their gradient with respect to a loss. All possible realizations of f through its weights draw a family N which expressivity is crucial for the success of the training. The common points between f and other objects of N define what is called a neural network *architecture*. That is

We present gradient based learning more formally in what follows.

Definition 22. Architecture Let $f \in \mathcal{N}$ with weights $(\theta_k)_k \in$.

Gradient descent

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). In order to be trained, f is parameterized with initial weights that characterize its linear parts. These weights are modified step by step in the opposite direction of their gradient with respect to a loss.

Architecture

All possible realizations of f through its weights draw a family N which expressivity is crucial for the success of the training. The common points between f and other objects of N define what is called a neural network *architecture*. TODO: refactor

Definition 23. 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:

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

where α can be a scalar or a vector, $\cdot$ can denote outer or pointwise product, and R is a regularizer. They depend on the optimization algorithm.
TODO: examples of optimization

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:

$$\vec{\nabla}_{\theta_k} = J_{\theta_k}(x_k) \vec{\nabla}_{x_k} \tag{4}$$

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

Obtaining,

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

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 $\mathcal{L}$, 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:

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

$$J_{x'_k}(h(x'_k)) = I\, h'(x'_k)$$

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

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.

TODO: Overfitting remark

1.2.5 Examples of layer

Definition 24. 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 25. 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.

Definition 26. 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 27. 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.

Definition 28. 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 27 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.

Affect on tensor shape

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 29. 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 27, 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 1.2.2. On the other hand, they increase memory and computational footprints.

Proposition 30. Connectivity matrix of a convolution with padding

A convolutional layer with padding (g, h) is equivalently defined as W_g being a $n_i \times n_o$ block matrix such that its blocks are Toeplitz matrices.

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

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

$$= \sum_{i_1=j_1}^{j_1+n_1^{(w)}-1} \cdots \sum_{i_n=j_n}^{j_n+n_n^{(w)}-1} x[i_1,\ldots,i_n]\, w[j_1+n_1^{(w)}-i_1,\ldots,j_n+n_n^{(w)}-i_n]$$

$$= \sum_{i_1=1}^{n_1^{(x)}} \cdots \sum_{i_n=1}^{n_n^{(x)}} x[i_1,\ldots,i_n]\, \widetilde{w}[i_1,j_1,\ldots,i_n,j_n]$$

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

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

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 $\{(j_1,\ldots,j_n), \forall t, 1 \le j_t \le n_t^{(y)}\}$. 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 $g_i(i_1,i_2,\ldots,i_n) = g_j(i_1-o_1, i_2-o_2, \ldots, i_n-o_n)$ defined if and only if $\forall t, 1+o_t \le i_t \le n_t^{(y)}$, 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

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

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. The former proof makes clear that the result doesn't 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.

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 don't (LeCun and Bengio, 1995). A more thorough comparison and explanation of their assets will be discussed in Section 2.1.3.

Definition 31. Stride

A convolutional layer with *stride* is a convolutional layer that computes

strided convolutions (with stride > 1) instead of convolutions.

Definition 32. 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.

Downscaling
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.
TODO: below

1.2.6 Examples of regularization

Overfitting
TODO:

A layer with *dropout* (g, h) is such that $h = h_1 \circ h_2$, where (g, h_2) is a layer and h_1 is a dropout operation (Srivastava et al., 2014). When dropout is used, a certain number of neurons are randomly set to zero during the training phase, compensated at test time by scaling down the whole layer. This is done to prevent overfitting.

1.2.7 Examples of architecture

TODO: rephrase

A multilayer perceptron (MLP) (Hornik et al., 1989) is a neural network composed of only dense layers. A convolutional neural network (CNN) (LeCun et al., 1998a) is a neural network composed of convolutional layers.

Neural networks are commonly used for machine learning tasks. For example, to perform supervised classification, we usually add a dense output layer $s = (g_{L+1}, h_{L+1})$ with as many neurons as classes. We measure the error between an output and its expected output with a discriminative loss function $\mathcal{L}$. During the training phase, the weights of the network are adapted for the classification task based on the errors that are back-propagated (Hornik et al., 1989) via the chain rule and according to a chosen optimization algorithm (*e.g.* Bottou, 2010).

1.3 Graphs and signals

1.3.1 Basic definitions

1.3.1.1 Graphs

Definition 33. Graph
A *graph* G is defined as a couple of sets $\langle V, E \rangle$ where V is the set of *vertices*, also called *nodes*, and $E \subseteq \binom{V}{2}$ is the set of *edges*. For all $u, v \in V$ we define the relation $u \sim v \Leftrightarrow \{u, v\} \in E$. Unless stated otherwise, we will consider only *weighted* graphs *i.e.* each graph G is associated with a weight mapping $w : E \to \mathbb{R}^*$.

Figure 1.2 illustrates an example of a graph. Note that we employ interchangeably the terms *vertex* and *node*.

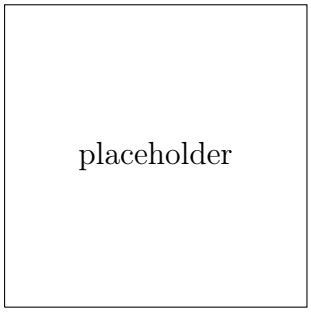

Figure 1.2: Example of a graph

Remark. According to this definition, we consider simple graphs *i.e.* no two edges share the same set of vertices and there is no self-loop.

Definition 34. Path
A *path* of length $n \in \mathbb{N}$ in a graph $G = \langle V, E \rangle$ is a sequence $(v_1 v_2 \cdots v_n)$ in V such that $\forall i, v_i \sim v_{i+1}$.

Remark. Our definition of graphs admit no self-loop so $\forall i, v_i \neq v_{i+1}$

Definition 35. Order
The order of a graph $G = \langle V, E \rangle$ is define as $\text{order}(G) = |V| \in \mathbb{N} \cup \{+\infty\}$

Definition 36. Adjacency matrix
The *adjacency matrix* of a finite graph $G = \langle V, E \rangle$ of order n, is a $n \times n$ real-valued matrix A associated to an indexing of $V = \{v_1, v_2, \dots v_n\}$, such that

$$A[i,j] = \begin{cases} w\left(\{v_i, v_j\}\right) & \text{if } v_i \sim v_j \\ 0 & \text{otherwise} \end{cases}$$

Definition 37. Degree
The *degree* of a vertex $v \in V$ of a graph $G = \langle V, E \rangle$ is defined as $\deg(v) = |\{u \in V, u \sim v\}| \in \mathbb{N} \cup \{+\infty\}$.
The *degree* of the graph G is defined as $\deg(G) = \max_{v \in V}\{\deg(v)\}$.
A graph is said to be *regular* if deg is constant on the vertices.

Definition 38. Degree matrix
The *degree matrix* of a finite graph $G = \langle V, E \rangle$ of order n, is the diagonal matrix D, associated to an indexing of $V = \{v_1, v_2, \dots v_n\}$, such that $D = \text{diag}(\deg(v_1), \deg(v_2), \dots, \deg(v_n))$.

Definition 39. Laplacian matrix
The *laplacian matrix* of a graph $G = \langle V, E \rangle$ of order n, associated to an indexing of $V = \{v_1, v_2, \dots v_n\}$, is defined as $L = D - A$, where D is the degree matrix and A is the adjacency matrix.

Definition 40. Digraph
A digraph is an oriented graph *i.e.* $E \subseteq V \times V - \{(v, v), v \in V\}$. Contrary to a graph, the weight mapping w, the relation $\sim$, the adjacency matrix A, and the laplacian matrix L are not symmetric. Notions defined on graphs naturally extends to digraphs where possible.

Definition 41. Bipartite graph
A *bipartite graph* is a triplet of sets $\langle V^{(1)}, V^{(2)}, E\rangle$, where $V^{(1)}$ and $V^{(2)}$ are sets of vertices, $V^{(1)} \cap V^{(2)} \neq \emptyset$, and $E \subseteq V^{(1)} \times V^{(2)}$. It is associated with a weight mapping $w : E \to \mathbb{R}^*$. Its adjacency matrix A is associated to indexings of $V^{(1)} = \{v_1^{(1)}, v_2^{(1)}, \dots v_n^{(1)}\}$ and $V^{(2)} = \{v_1^{(2)}, v_2^{(2)}, \dots v_n^{(2)}\}$, such that

$$A[i,j] = \begin{cases} w\left((v_i^{(1)}, v_j^{(2)})\right) & \text{if } (v_i^{(1)}, v_j^{(2)}) \in E \\ 0 & \text{otherwise} \end{cases}$$

Definition 42. Induced subgraph
The *subgraph* $\widetilde{G} = \langle \widetilde{V}, \widetilde{E}\rangle$ of a graph $G = \langle V, E\rangle$, *induced* by $\widetilde{V} \subseteq V$, is such that $\forall (u,v) \in \widetilde{V}^2, u \overset{\widetilde{G}}{\sim} v \Leftrightarrow u \overset{G}{\sim} v$.

TODO: subgraph (including direct or undirected)

Definition 43. Grid graph
A *grid graph* $G = \langle V, E\rangle$ is such that $V \cong \mathbb{Z}^2$, $v_1 \sim v_2 \Rightarrow \|v_2 - v_1\|_\infty \in \{0,1\}$ and either one of the following is true:

$$\begin{cases} (i_1,j_1) \sim (i_2,j_2) \Leftrightarrow |i_2 - i_1| \,\text{XOR}\, |j_2 - j_1| & \text{(4 neighbours)} \\ (i_1,j_1) \sim (i_2,j_2) \Leftrightarrow |i_2 - i_1| \,\text{AND}\, |j_2 - j_1| & \text{(4 neighbours)} \\ (i_1,j_1) \sim (i_2,j_2) \Leftrightarrow |i_2 - i_1| \,\text{OR}\, |j_2 - j_1| & \text{(8 neighbours)} \end{cases}$$

A *(rectangular) grid graph* of size $n \times m$ is the subgraph of a grid graph induced by $[\![1,n]\!] \times [\![1,m]\!]$.
A *square grid graph* is a rectangular grid graph of square size.

1.3.1.2 (Real-valued) Signals

Definition 44. Signal space
The *signal space* $\mathcal{S}(V)$, over the set V, is the linear space of real-valued

functions defined on V.

We have $\dim(\mathcal{S}(V)) = |V| \in \mathbb{N} \cup \{+\infty\}$.

Remark. In particular, a vector space, and more generally a tensor space, are finite-dimensional signal spaces over any of their bases.

Definition 45. Signal
A *signal* over V, $s \in \mathcal{S}(V)$, is a function $s : V \to \mathbb{R}$.
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 a 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 $\text{supp}(s) = \{v \in V, s[v] \neq 0\}$.

Definition 46. Graph signal
A *graph signal* over G is a signal over its vertex set. We denote by $\mathcal{S}(G)$ the graph signal space.

We have $\dim(\mathcal{S}(G)) = \text{order}(G) \in \mathbb{N} \cup \{+\infty\}$.

Definition 47. Underlying structure
An *(underlying) structure* of a signal s over a set V, is a graph G with vertex set V.

Examples
When there is a unique clear underlying structure, we say that it is *the* underlying structure. For example, images are compactly supported signals over $\mathbb{Z}^2$ and their underlying structure is a rectangular grid graph. Time series are signals over $\mathbb{N}$ and their underlying structure is a line graph. The underlying structure of a graph signal is obviously the graph itself.

1.3.2 Graphs in deep learning

TODO: below

We come across the notion of graphs several times in deep learning:

- Connections between two layers of a deep learning model can be represented as a bipartite graph, the *connectivity graph*. It encodes how the information is propagated through a layer to another. See Section 1.3.2.1.
- Neural architectures can be represented by a graph. In particular, a computation graph is used by deep learning programming languages to keep track of the dependencies between layers of a deep learning model, in order to compute forward and back-propagation. See Section 1.3.2.2.
- A graph can represent the underlying structure of an object (often a vector or a signal). The nodes represent its features, and the edges represent some structural property. See Section 1.3.2.3.
- Datasets can also be graph-structured. The nodes represent the objects of the dataset, and its edge represent some sort of relation between them. See Section 1.3.2.4.

1.3.2.1 Connectivity graph

A Connectivity graph is the bipartite graph whose adjacency matrix is the connectivity matrix of a layer of neurons. Formally, given a linear part of a layer, let $\mathbf{x}$ and $\mathbf{y}$ be the input and output signals, n the size of the set of input neurons $N = \{u_1, u_2, \dots, u_n\}$, and m the size of the set of output neurons $M = \{v_1, v_2, \dots, v_m\}$. This layer implements the equation $y = \Theta x$ where Θ is a $n \times m$ matrix.

Definition 48. The *connectivity graph* $G = (V, E)$ is defined such that $V = N \cup M$ and $E = \{(u_i, v_j) \in N \times M, \Theta_{ij} \neq 0\}$.

I.e. the connectivity graph is obtained by drawing an edge between neurons for which $\Theta_{ij} \neq 0$. For instance, in the special case of a complete bipartite graph, we would obtain a dense layer. Connectivity graphs are especially useful to represent partially connected layers, for which most of the Θ_{ij} are

0. For example, in the case of layers characterized by a small local receptive field, the connectivity graph would be sparse, and output neurons would be connected to a set of input neurons that corresponds to features that are close together in the input space. Figure 1.3 depicts some examples.

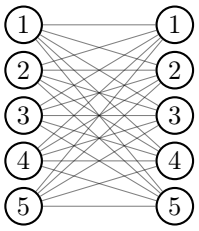

Figure 1.3: Examples

TODO: Figure 1.3. It's just a placeholder right now

Connectivity graphs also allow to graphically modelize how weights are tied in a neural layer. Let's suppose the $\Theta_i j$ are taking their values only into the finite set $K = \{w_1, w_2, \ldots, w_\kappa\}$ of size κ, which we will refer to as the *kernel* of *weights*. Then we can define a labelling of the edges $s : E \to K$. s is called the *weight sharing scheme* of the layer. This layer can then be formulated as $\forall v \in M, y_v = \sum_{u \in N, (u,v) \in E} w_{s(u,v)} x_u$. Figure 3.3.2 depicts the connectivity graph of a 1-d convolution layer and its weight sharing scheme.

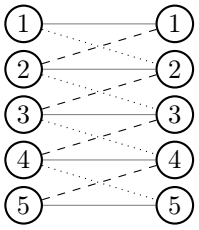

Figure 1.4: Depiction of a 1D-convolutional layer and its weight sharing scheme.

TODO: Add weight sharing scheme in Figure 3.3.2

1.3.2.2 Computation graph

1.3.2.3 Underlying graph structure and signals

1.3.2.4 Graph-structured dataset

Chapter 2

Convolutions on graph domains

Contents

Introduction

Defining a convolution of signals over graph domains is a challenging problem. Obviously, if the graph is not a grid graph there exists no natural definition. We first analyse the reasons why the classical convolution operator is useful in deep learning, and give a characterization. Then we will search for domains onto which a convolution with this characteristics can be naturally obtained. This will lead us to put our interest on representation theory and convolutions defined on algebraic structures, such as groups, in order to transfer its construction on vertex domains of graphs, agnostically of the edge set. Then, we will introduce the role of the edge sets and how it should influence the contruction. This will provide us with some particular classes of graphs for which we will obtain a natural construction with respect to the wanted characteristics. Finally, we study how we can loosen them to adapt the construction to more irregular domains.

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 49. 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 50. 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-action* operator $x \mapsto x * w$ from the *left-action* 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 w-weighted aggregation of entries of x in a neighbourhood of (a, b), called the *local receptive field*.

Commutativity
The 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 Proposition 80.

2.1.2 Characterization on grid graphs

Let's recall first what is a transformation, and how it acts on signals.

Definition 51. 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 bijective transformations is denoted $\Phi^*(V) \subset \Phi(V)$.
In case $f \in \Phi^*(V)$, it can act on $\mathcal{S}(V)$ through the linear operator $L_f \in \mathcal{L}(\mathcal{S}(V))$ defined as:

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

i.e. an entry of a transformed signal is obtained by doing a lookup of the entry of the original signal.
In case $f \notin \Phi^*(V)$, we can still define $L_f \in \mathcal{L}(\mathcal{S}(V))$, however we need to linearly aggregate the entries on the fibers:

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

where AGGREGATE can be for example the sum, the average, or the max, and $\text{AGGREGATE}(\emptyset) = 0$.

We also recall the formalism of translations.

Definition 52. 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

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

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

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

For any set E, we denote by $\mathcal{T}(E)$ its translations if they are defined.

The next proposition 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).

Proposition 53. 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:

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

$$\forall s \in \mathcal{S}(\mathbb{Z}^2), \forall (\alpha, \beta) \in \mathbb{Z}^2, \forall (a, b) \in \mathbb{Z}^2,$$

$$\begin{aligned} f_w \circ t_{\alpha,\beta}(s)[a,b] &= \sum_i \sum_j t_{\alpha,\beta}(s)[i,j]\, w[a-i, b-j] \\ &= \sum_i \sum_j s[i-\alpha, j-\beta]\, w[a-i, b-j] \\ &= \sum_{i'} \sum_{j'} s[i', j']\, w[a-i'-\alpha, b-j'-\beta] \qquad (13) \\ &= f_w(s)[a-\alpha, b-\beta] \\ &= t_{\alpha,\beta} \circ f_w(s)[a,b] \qquad \text{(equivariance)} \end{aligned}$$

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

Let $f \in \mathcal{L}(\mathcal{S}(\mathbb{Z}^2))$, $s \in \mathcal{S}(\mathbb{Z}^2)$. We suppose that f commutes with translations. Recall that s can be linearly decomposed on the infinite family of dirac signals:

$$s = \sum_i \sum_j s[i,j]\, \delta_{i,j}, \text{ where } \delta_{i,j}[x,y] = \begin{cases} 1 & \text{if } (x,y) = (i,j) \\ 0 & \text{otherwise} \end{cases}$$

By linearity of f and then equivariance to translations:

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

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

$$\begin{aligned} \forall (a,b) \in \mathbb{Z}^2, f(s)[a,b] &= \sum_i \sum_j s[i,j]\, t_{i,j}(w)[a,b] \qquad (14) \\ &= \sum_i \sum_j s[i,j]\, w[a-i, b-j] \end{aligned}$$

$$i.e.\ f(s) = s * w$$

□

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 Proposition 80, 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 with strictly no 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.

Methodology for extending to general graphs

Hence, in our construction, we will try to preserve the characterization from Proposition 80 as it is mostly the reason why they are successful in deep

learning. Note that the reduction of parameters compared to a dense layer is also a consequence of this characterization.

2.2 Construction from the vertex set

2.2.1 Introduction

As Proposition 80 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 signals 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 Proposition 80 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 a subgroup of the transformations of this set).

- Can a group domain draw an underlying graph structure ? Is the group convolution naturally defined on this class of graphs ?

We first recall the notion of group and group convolution.

Definition 54. Group
A group $\mathcal{G}$ is a set equipped with a closed, associative and invertible composition law that admits a unique left-right identity element.

Definition 55. Group convolution I
Let a group $\mathcal{G}$, the group convolution I between two signals s_1 and $s_2 \in \mathcal{S}(\mathcal{G})$ is defined as:

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

provided at least one of the signals has finite support if $\mathcal{G}$ is not finite.

2.2.2 Steered construction from groups

For a graph $G = \langle V, E \rangle$ and a subgroup $\Gamma \subset \Phi^*(V)$ or its invertible transformations, Definition 55 is applicable for $\mathcal{S}(\Gamma)$, but not for $\mathcal{S}(V)$ as V is not a group. Nonetheless, our point here is that we will use the group convolution on $\mathcal{S}(\Gamma)$ to construct the convolutions on $\mathcal{S}(V)$.
For now, let's assume Γ is in one-to-one correspondence with V, and let's define an isomorphism φ from Γ to V, to equip V with a group structure. 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 $\widetilde{\varphi}(\delta_g) = \delta_{\varphi(g)}$ is also an isomorphism. Hence, V and $\mathcal{S}(V)$ would inherit the same inherent structural properties as Γ and $\mathcal{S}(\Gamma)$, in addition of being related in the same way. 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 2.1.

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

Figure 2.1: Commutative diagram between sets

Then 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.

Lemma 56. 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.

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

□

Hence, we can steer the definition of the group convolution from $\mathcal{S}(\Gamma)$ to $\mathcal{S}(V)$ as follows:

Definition 57. Group convolution II
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$. The group convolution II between two signals s_1 and $s_2 \in \mathcal{S}(V)$ is defined as:

$$\begin{aligned}\forall u \in V, (s_1 *_{\text{II}} s_2)[u] &= \sum_{v\in V} s_1[v]\, s_2[\varphi(g_v^{-1} g_u)] \\ &= \sum_{\substack{(a,b)\in V^2 \\ s.t.\ g_a g_b = g_u}} s_1[a]\, s_2[b]\end{aligned}$$

Lemma 58. Relation between group convolution I and II
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$,

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

Proof. Using Lemma 56,

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

□

Proposition 59. Equivariance to $\varphi(\Gamma)$
With Definition 57, convolution operators acting on the right of $\mathcal{S}(V)$ are equivariant to $\varphi(\Gamma)$ *i.e.*

$$\exists w \in \mathcal{S}(V), f = . *_{\text{II}} w \Rightarrow \forall v \in V, f \circ \varphi(g_v) = \varphi(g_v) \circ f$$

Proof.

$$\begin{aligned}&\forall s \in \mathcal{S}(V), \forall u \in V, \forall v \in V, \\ &(f_w \circ \varphi(g_u))(s)[v] = \sum_{\substack{(a,b) \in V^2 \\ s.t.\ g_a g_b = g_v}} \varphi(g_u)(s)[a]\, w[b] \\ &= \sum_{\substack{(a,b) \in V^2 \\ s.t.\ g_a g_b = g_v}} s[\varphi(g_u)^{-1}(a)]\, w[b] \\ &= \sum_{\substack{(a,b) \in V^2 \\ s.t.\ g_{\varphi(g_u)(a)} g_b = g_v}} s[a]\, w[b]\end{aligned}$$

Because φ is an isomorphism, its inverse $c \mapsto g_c$ is also an isomorphism and

so $g_{\varphi(g_u)(a)}g_b = g_v \Leftrightarrow g_a g_b = g_{\varphi(g_u)^{-1}(v)}$. So we have both:

$$\begin{aligned}(f_w \circ \varphi(g_u))(s)[v] &= \sum_{\substack{(a,b)\in V^2 \\ s.t.\ g_a g_b = g_{\varphi(g_u)^{-1}(v)}}} s[a]\, w[b] \\ &= s *_{\text{II}} w[\varphi(g_u)^{-1}(v)] \\ &= (\varphi(g_u) \circ f_w)(s)[v]\end{aligned}$$

□

Remark. Note that convolution operators of the form $f_w = . *_{\text{I}} w$ are also equivariant to $\mathcal{G}$, but the proposition and the proof are omitted as they are similar to the latter.

In fact, both group convolutions are the same as the latter one borrows the algebraic structure of the first one. Thus we only obtain equivariance to $\varphi(\Gamma)$, which is actually V equipped with the group structure of Γ mirrored via φ, and the converse don't hold. To obtain equivariance to Γ (and its converse), we need to take into account the fact that it contains bijective transformations of V.
Hence, note that $g \in \Gamma$ can act on both Γ through the left multiplication and on V as being an object of $\Phi^*(V)$. This ambivalence can be seen on a commutative diagram, see Figure 2.2.

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

Figure 2.2: Commutative diagram. All arrows except for the one labeled with (P) are always True.

For (P) to be true means that φ is an equivariant map between groups *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 (in the isomorphic sense) and we denote $\Gamma \equiv V$.

Definition 60. Equivariant map
A homomorphism φ from a group of permutations $\mathcal{G}$ and a group $\mathcal{H}$ is said to be an *equivariant map* if

$$\forall g, h \in \mathcal{G}, g(\varphi(h)) = \varphi(gh)$$

In our case 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.

Definition 61. φ-Equivalence
A subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$, is said to be *$\varphi$-equivalent* if φ is an equivariant map *i.e.* if it verifies the property:

$$\forall v, u \in V, g_v(u) = \varphi(g_v g_u) \tag{P}$$

In that case we denote $\Gamma \overset{\varphi}{\equiv} V$.

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

Definition 62. Group convolution III
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $\Gamma \overset{\varphi}{\cong} V$. The group convolution III between two signals s_1 and $s_2 \in \mathcal{S}(V)$ is defined as:

$$s_1 *_{\text{III}} s_2 = \sum_{v \in V} s_1[v]\, g_v(s_2) \tag{15}$$

$$= \sum_{g \in \Gamma} s_1[\varphi(g)]\, g(s_2) \tag{16}$$

The two expressions differ on the domain upon which the summation is done. The expression (15) put the emphasis on each vertex and its action, whereas the expression (16) emphasizes on each object of Γ.

Lemma 63. Relation with group convolution II
$\Gamma \overset{\varphi}{\equiv} V \Leftrightarrow *_{\text{II}} = *_{\text{III}}$

Proof.

$$\begin{aligned}
&\forall s_1, s_2 \in \mathcal{S}(V), \\
&\quad s_1 *_{\text{II}} s_2 = s_1 *_{\text{III}} s_2 \\
&\quad \Leftrightarrow \forall u \in V, \sum_{v \in V} s_1[v]\, s_2[\varphi(g_v^{-1} g_u)] = \sum_{v \in V} s_1[v]\, s_2[g_v^{-1}(u)] \qquad (17)
\end{aligned}$$

Hence, the direct sense is obtained by applying (P).
For the converse, given $u, v \in V$, we first realize (17) 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 (P) with the one-to-one correspondence $v := v^{-1}$, where $v^{-1} = \varphi(g_v^{-1})$ and using the fact that φ and φ^{-1} are isomorphisms. □

We can then coin it as φ-convolution.

Definition 64. φ-convolution
Let $\varphi \in \text{ISO}(\Gamma, V), \Gamma \overset{\varphi}{\equiv} V$, the *$\varphi$-convolution* between two signals s_1 and $s_2 \in \mathcal{S}(V)$ is defined as:

$$s_1 *_{\varphi} s_2 = s_1 *_{\text{II}} s_2 = s_1 *_{\text{III}} s_2$$

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

Proposition 65. Characterization by right-action equivariance to Γ
If Γ is φ-equivalent, the class of linear transformations that are equivariant

to Γ is exactly the class of φ-convolution operators acting on the right of $\mathcal{S}(V)$ *i.e.*

$$\text{If } \Gamma \overset{\varphi}{\equiv} V,$$
$$\exists w \in \mathcal{S}(V), f = . *_\varphi w \Leftrightarrow \begin{cases} f \in \mathcal{L}(\mathcal{S}(V)) \\ \forall g \in \Gamma, f \circ g = g \circ f \end{cases}$$

Proof. 1. From left to right:

In the following equations, (18) is obtained by definition, (19) is obtained because left multiplication in a group is bijective, and (20) is obtained because of (P).

$$\begin{aligned} &\forall g \in \Gamma, \forall s \in \mathcal{S}(V), \\ &\qquad f_w \circ g(s) = \sum_{h \in \Gamma} g(s)[\varphi(h)]\, h(w) && (18) \\ &\qquad\qquad = \sum_{h \in \Gamma} g(s)[\varphi(gh)]\, gh(w) && (19) \\ &\qquad\qquad = \sum_{h \in \Gamma} g(s)[g(\varphi(h))]\, gh(w) && (20) \\ &\qquad\qquad = \sum_{h \in \Gamma} s[\varphi(h)]\, gh(w) \\ &\qquad\qquad = \sum_{h \in \Gamma} s[\varphi(h)]\, h(w)[g^{-1}(.)] \\ &\qquad\qquad = f_w(s)[g^{-1}(.)] \\ &\qquad\qquad = g \circ f_w(s) \end{aligned}$$

Of course, we also have that f_w is linear.

2. From right to left:

Let $f \in \mathcal{L}(\mathcal{S}(V)), s \in \mathcal{S}(V)$. By linearity of f, we distribute $f(s)$ on

the family of dirac signals:

$$f(s) = \sum_{v \in V} s[v] f(\delta_v) \tag{21}$$

Thanks to (P), we have that:

$$\begin{aligned} & g_v(\varphi(\mathrm{Id})) = \varphi(g_v\,\mathrm{Id}) = v \\ & \text{So, } v = u \Leftrightarrow \varphi(\mathrm{Id}) = g_v^{-1}(u) \\ & \text{So, } \delta_v = g_v(\delta_{\varphi(\mathrm{Id})}) \end{aligned}$$

By denoting $w = f(\delta_{\varphi(\mathrm{Id})})$, and using the hypothesis of equivariance, we obtain from (21) that:

$$\begin{aligned} f(s) &= \sum_{v \in V} s[v]\, f \circ g_v(\delta_{\varphi(\mathrm{Id})}) \\ &= \sum_{v \in V} s[v]\, g_v \circ f(\delta_{\varphi(\mathrm{Id})}) \\ &= \sum_{v \in V} s[v]\, g_v(w) \\ &= s *_\varphi w \end{aligned}$$

□

Construction of φ-convolutions on vertex domains

Proposition 65 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 transformations of V, that is equivalent to V. The choice of Γ can be done with respect to some graph symmetries related to E. This will be discussed in more details in Section 2.3.

Exposure of φ
This construction relies on exposing an equivariant map φ between Γ and V. In the next subsection, we show that in cases where Γ is abelian, we even need not expose φ to preserve the characterization.

2.2.3 Mixed domain formulation

From (16), 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 the isomorphisms φ.

Definition 66. Mixed domain convolution
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $V \cong \Gamma$. 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:

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

We coin 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 67. Relation with group convolution III
$\forall \varphi \in \text{ISO}(\Gamma, V), \forall (r, s) \in \mathcal{S}(\Gamma) \times \mathcal{S}(V)$,

$$r *_{\text{M}} s = \varphi(r) *_{\text{III}} s$$

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

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

Where $\overset{(\diamond)}{=}$ comes from Lemma 56. □

In other words, $*_{\text{M}}$ is a convenient reformulation of $*_{\text{III}}$ which does not depend on a particular φ.

Lemma 68. Relation with group convolution I, II and φ-convolution
Let $\varphi \in \text{ISO}(\Gamma, V), (r, s) \in \mathcal{S}(\Gamma) \times \mathcal{S}(V)$, we have:

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

Proof. On one hand, Lemma 67 gives $r *_{\text{M}} s = \varphi(r) *_{\text{III}} s$. On the other hand, Lemma 58 gives $\forall v \in V, (r *_{\text{I}} \varphi^{-1}(s))[g_v] = (\varphi(r) *_{\text{II}} s)[v]$. Then Lemma 63 concludes. □

Remark. The converse sense is meaningful because it justifies that when the M-convolution is employed, the property $\Gamma \equiv V$ underlies, without the need of expliciting φ.

From M-convolution, we can derive operators acting on the left of $\mathcal{S}(V)$, of the form $s \mapsto w *_{\text{M}} s$, parameterized by $w \in \mathcal{S}(\Gamma)$. In particular, these operators would be relevant as layers of neural networks. On the contrary, derived operators acting on the right such as $r \mapsto r *_{\text{M}} w$ wouldn't make sense with this formulation as they would make φ resurface. However, the equivariance to Γ incurring from Lemma 67 and Proposition 65 only holds for operators acting on the right. So we need to intertwine an abelian condition as follows. This is also a good excuse to see the influence of abelianity.

Proposition 69. Equivariance to Γ through left action
Let a subgroup $\Gamma \subset \Phi^*(V)$ such that $V \cong \Gamma$. Γ is abelian, if and only if, M-convolution operators acting on the left of $\mathcal{S}(V)$ are equivariant to it *i.e.*

$$\forall g, h \in \Gamma, gh = hg \Leftrightarrow \forall w, g \in \Gamma, w *_{\text{M}} g(.) = g \circ (w *_{\text{M}} .)$$

Proof. Let $w, g \in \Gamma$, and define $f_w : s \mapsto w *_{\text{M}} s$. In the following expressions, Γ is abelian if and only if (22) and (23) are equal (the converse is obtained by particularizing on well chosen signals):

$$\begin{aligned} f_w \circ g(s) &= \sum_{h \in \Gamma} w[h]\, hg(s) && (22) \\ &= \sum_{h \in \Gamma} w[h]\, gh(s) && (23) \\ &= \sum_{h \in \Gamma} w[h]\, h(s)[g^{-1}(.)] \\ &= (w *_{\text{M}} s)[g^{-1}(.)] \\ &= g \circ f_w(s) \end{aligned}$$

□

Remark. Similarly, $*_\varphi$ is also equivariant to Γ through left action if and only if Γ is abelian, as a consequence of being commutative if and only if Γ is abelian. On the contrary, note that commutativity of $*_{\text{M}}$ doesn't make sense.

Corrolary 70. Characterization by left-action equivariance to Γ
Let $\Gamma \equiv V$. If Γ is abelian, the class of linear transformations that are equivariant to Γ is exactly the class of M-convolution operators acting on the left of $\mathcal{S}(V)$ *i.e.*

$$\begin{aligned} &\text{If } \Gamma \equiv V \text{ and } \Gamma \text{ is abelian,} \\ &\exists w \in \mathcal{S}(\Gamma), f = w *_{\text{M}} . \Leftrightarrow \begin{cases} f \in \mathcal{L}(\mathcal{S}(V)) \\ \forall g \in \Gamma, f \circ g = g \circ f \end{cases} \end{aligned}$$

Proof. By picking φ such that $\Gamma \overset{\varphi}{\equiv} V$ and relating $*_{\text{M}}$ to $*_\varphi$. □

Depending on the applications, we will build upon either $*_\varphi$ or $*_{\text{M}}$ if the abelian condition can be verified.

2.3 Construction with the edge set

2.3.1 Introduction

The constructions from the previous section involve the vertex set V and depend on Γ, a subgroup of the set of invertible transformations on V. Therefore, it looks natural to try to relate the edge set and Γ.
There are two approaches. 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.
TODO: introduction to be developped more

2.3.2 Edge-constrained transformations

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), under the condition that Γ is generated by a set of edge-constrained transformations ?

Definition 71. 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 Φ_{EC} and Φ^*_{EC} the sets of EC and invertible EC transformations.

Remark. Note that Φ^*_{EC} is not a group, thus why we are only interested in groups generated by its subsets.

This leads us to consider Cayley graphs (Cayley, 1878; Wikipedia, 2018a).

Definition 72. 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:

$$u \to v \Leftrightarrow \exists g \in \mathcal{U}, ga = b$$

Also, if Γ is abelian, we call it an *abelian Cayley graph*.

Remark. Note that for compatibility with the functional notation that we use, we define Cayley graphs with $ga = b$ instead of $ag = b$.

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

Proposition 73. Characterization by Cayley subgraph isomorphism
Let a graph $G = \langle V, E \rangle$, then:

1. G admits an (EC) φ-convolution if and only if it contains a subgraph isomorph to a Cayley graph

2. G admits an (EC) M-convolution if and only if it contains a subgraph isomorph to an abelian Cayley graph

Proof. We show the result in the non-abelian case as the proof for the abelian case is similar.

1. From left to right:

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 Cayley graph $\vec{\Gamma} = \langle \Gamma, \mathcal{E} \rangle$, generated by $\mathcal{U}$. Let ψ be a graph ismorphism from V_S to Γ. $\forall g \in \mathcal{U}$, we define its action on V_s as $L_g(u) = w \Leftrightarrow g\psi(u) = \psi(w)$. As ψ is a graph isomorphism, the invertible transformations $L_{\mathcal{U}} = \{L_g, g \in \mathcal{U}\}$ are (EC). Then, by induction, whenever the actions L_g and L_h are defined, we define the action of gh as $L_{gh} = L_g \circ L_h$. As $\langle \mathcal{U} \rangle = \Gamma$, L_g is defined for all $g \in \Gamma$. Moreover, the map $g \mapsto L_g$ is injective because $L_g = L_h \Rightarrow \forall u \in V, L_g(u) = L_h(u) \Rightarrow \forall u \in V, g\psi(u) = h\psi(u) \Rightarrow g = h$, so all the actions are well defined (*i.e.* whenever L_{gh} was already defined via $xy = gh$, it is redefined similarly).

TODO: equivariant map, via identity rooting

□

TODO: below to reword

TODO: operator and characterization
TODO: which graph is a Cayley graph ?

2.3.3 Construction on graph groupoids

TODO: work in progress

On graphs, we notice that the property (P) can be realized by transformations acting on edges. However, unless the graph is complete, these actions can't be composed everywhere to form another edge constrained action. The algebraic structure that posesses the same kind of properties than a group except that its composition law is not defined everywhere is called a groupoid. The following definitions clarify our discussion.

Definition 74. Groupoid
A groupoid is a set equiped with a closed partial composition law, a unique identity element, and every unique inverses.

Remark. We use the convention than left and right inverses must be the same.

Definition 75. Graph groupoid
The *groupoid* $\mathcal{P}(G)$ of a graph $G = \langle V, E \rangle$ is the set of its paths equiped with:

1. two maps ψ and φ that respectively map a path to its first and last element,
2. a closed partial composition law gh defined if and only if $\psi(g) = \varphi(h)$, which concatenates g behind h and ~~removes adjacent duplicate vertices~~ to rewrite,
3. an inverse operator $^{-1}$ which maps a path to its reverse,
4. an identity element Id which is the path of length 0.

Remark. Recall from Definition 34 that a path can't contain adjacent duplicates.

Remark. Note that even though the composite path gh has elements of h before those of g we write gh instead of hg because we'll need the left operand to act on the right one through functional notation $g(h)$.

Definition 76. Graph k-groupoid
The *k-groupoid* $\mathcal{P}_k(G)$ of a graph $G = \langle V, E \rangle$, for $k \in \mathbb{N}^*$, is the groupoid obtained by restricting $\mathcal{P}(G)$ to paths of length at most k (the definition domain of its composition law is also further restricted by the length of the resulting paths in $\mathcal{P}(G)$).

Definition 77. k-Groupoid convolution
Let a graph $G = \langle V, E \rangle$. Let a subgroupoid $\Gamma \subseteq \mathcal{P}_k(G)$. The k-groupoid convolution between two signals s_1 and $s_2 \in \mathcal{S}(\Gamma)$ is defined as:

$$\begin{aligned}\forall h \in \Gamma, (s_1 * s_2)[h] &= \sum_{\substack{(a,b)\in\Gamma^2 \\ s.t.\ ab=h}} s_1[a]\, s_2[b] \\ &= \sum_{\substack{g\in\Gamma \\ s.t.\ \varphi(g)=\varphi(h)}} s_1[g]\, s_2[g^{-1}h] \\ &= \sum_{\substack{g\in\Gamma \\ s.t.\ \psi(g)=\psi(h)}} s_1[hg^{-1}]\, s_2[g]\end{aligned}$$

Claim 78. Path transformation
Let a graph $G = \langle V, E \rangle$. By identifying vertices with paths of length 1, a path $g \in \mathcal{P}(G)$ can act as a transformation on $v \in V$ through the composition law of $\mathcal{P}(G)$. Also note that $g(v) = g(v^{-1})$.

We can nom define the k-Groupoid convolution operator on $\mathcal{S}(G)$ by restriction of the second operand from $\mathcal{S}(\Gamma)$ to paths of length 1:

Definition 79. k-Groupoid convolution operator
Let a graph $G = \langle V, E \rangle$. Let a subgroupoid $\Gamma \subseteq \mathcal{P}_k(G)$. The k-groupoid convolution operator f_w with parameter $w \in \mathcal{S}(\Gamma)$ is defined as:

$$\forall s \in \mathcal{S}(\Gamma), \forall h \in \Gamma, f_w(s)[h] = (s * w)[h]$$

And when restricted to $\mathcal{S}(G)$ it is defined as:

$$\forall s \in \mathcal{S}(G), \forall v \in V, f_w(s)[v] = \sum_{\substack{g\in\Gamma \\ s.t.\ \psi(g)=v}} s[g(v)]\, w[g]$$

$$\forall s \in \mathcal{S}(G), \forall v \in V, f_w(s)[v] = \sum_{\substack{g\in\Gamma \\ s.t.\ \varphi(g)=v}} s[g]\, w[g^{-1}(v)]$$

Proposition 80. Groupoid equivariance to Γ
k-Groupoid convolution operators on $\mathcal{S}(G)$ are groupoid equivariant to Γ *i.e.*

$$\exists w \in \mathcal{S}(\Gamma), f = w * . \Rightarrow \forall v \in V, \forall g \in \Gamma s.t.\ \psi(g^{-1}) = v, f \circ g[v] = g \circ f[v]$$

$$g(h(v)) maybe false \tag{24}$$

Mini patron of todo:

- Equivariance to Γ holds, proof
- Converse of characterization does not hold yet, except on orbits
- property for it to hold
- relaxing one-to-one correspondence constraint but keeping other properties
- other avenue instead of property: should make use of edges to build a group structure
- ideal graph (lattice-regular)
- if group is too much then just groupoid structure from edges is enough

TODO: finish this section

2.3.4 To rename

Definition 81. Graph automorphisms
A graph automorphism of a graph $G = \langle V, E \rangle$ is a bijection in the vertex

domain $\phi : V \to V$ such that $\{u, v\} \in E \Leftrightarrow \{\phi(u), \phi(v)\} \in E$. We denote $\mathcal{A}(G)$ the group of automorphism on G.
We denote by $\mathcal{E}(\phi)$ the set of input-output mapping of ϕ, defined as $\mathcal{E}(\phi) = \{(x, y) \in V^2, \phi(x) = y\}$.
A graph automorphism ϕ is said to be *edge-constrained* (EC) if $\mathcal{E}(\phi) \subseteq E$. We denote $\mathcal{A}_{\text{EC}}(G)$ the set of edge-constrained automorphism on G.

Definition 82. Orthogonality
Two graph automorphisms ϕ_1 and ϕ_2 are said to be orthogonal, if and only if $\mathcal{E}(\phi_1) \cap \mathcal{E}(\phi_2) = \emptyset$, denoted $\phi_1 \perp \phi_2$. They are said to be aligned otherwise. Similarly, we define orthogonality of r automophisms as $\phi_1 \perp \cdots \perp \phi_r \Leftrightarrow \mathcal{E}(\phi_1) \cap \cdots \cap \mathcal{E}(\phi_r) = \emptyset$

2.3.5 Lattice-regular graph

Definition 83. Lattice-regular graph
A lattice-regular graph is a regular graph that admits r orthogonal edge-constrained automorphisms, where r is its degree.

2.4 Conclusion of chapter 2

TODO:

Chapter 3

Neural networks on graphs

Contents

Introduction

TODO:

3.1 Datasets

3.1.1 Supervised classification of graph signals

Image datasets
Distorded image datasets
Scrambled image datasets
Haxby
Pines fmri
20news

3.1.2 Semi-supervised classification of nodes

Cora
...

3.1.3 Supervised classification of graphs

Mutag
...

3.2 Related works

TODO: Presentation and note on older graph neural network models

3.2.1 Analysis of spectral techniques

Given a graph $G = \langle V, E \rangle$, spectral techniques are based on the graph Fourier transform (GFT) (Chung, 1996; Shuman et al., 2013), derived from the laplacian matrix of G. Spectral CNNs make use of a convolution defined as pointwise multiplication in the spectral domain defined by the GFT. Neural networks based on such methods were first introduced by Bruna et al., 2013. Henaff et al., 2015 later extended them to large scale classification tasks, and investigate the problem of estimating a suitable G from data.
Let a graph $G = \langle V, E \rangle$ of order n, with adjacency matrix A, degree matrix D, and let L denote either its normalized laplacian matrix or its unnormalized laplacian matrix. As a real symmetric matrix, L is diagonalized as $L = U^\mathsf{T} \Lambda U$, where columns of U form the eigenvectors basis $\mathcal{U}$ and $\Lambda = \operatorname{diag}(\lambda_1, \dots, \lambda_n)$ is a diagonal matrix of the corresponding eigenvalues.

Definition 84. Graph Fourier transform
The *GFT*, *w.r.t.* laplacian eigenbasis $\mathcal{U}$, of a signal defined on G, represented by a column vector $x \in \mathcal{S}(G)$, is defined as $\mathcal{F}(x) = Ux$.

The spectral convolution can then be constructed by similarity with the convolution theorem from classical Fourier analysis (Wikipedia, 2018b).

Definition 85. Spectral convolution
The spectral convolution between two signals $x_1, x_2 \in \mathcal{S}(G)$ is defined as

$$\begin{aligned} x_1 \star x_2 &= U^\mathsf{T}(Ux_1 \circ Ux_2) \\ &= U^\mathsf{T} \operatorname{diag}(Ux_1)\, Ux_2 \end{aligned}$$

where here $\circ$ denotes Hadamard product and $\forall v \in \mathbb{R}^n, \text{diag}(v) = \text{diag}(v[1], \ldots, v[n])$.

This gives rize to a class of convolution layers. In the following definition we reinterpret them with the formalism introduced in Section 1.2.

Definition 86. Spectral convolution layer with $\mathcal{O}(n)$ weights
A spectral convolution layer (g, h) is such that its connectivity matrix is of the form:

$$W_g = U^\mathsf{T} \text{diag}(\theta) U$$

where θ contains n learnable weights.

Remark. Spectral convolution layer are also extended with feature maps and input channels. In fact $W_g := W_g^{p,q}$ where $W_g^{p,q}$ is the (p, q) block corresponding to input channel p, and feature map q, of a larger connectivity matrix. But we omit the superscript for the sake of simplicity.

Such layers have two main drawbacks:

1. they produce $\mathcal{O}(n)$ weights instead of $\mathcal{O}(1)$ as in the case of classical two-dimensional convolutions,

2. connections of W_g do not depend on a locality notion based on the original adjacency matrix A.

Hence, Bruna et al., 2013 suggest to alleviate these issues with the following construction (reinterpreted with our formalism):

Definition 87. Spectral convolution layer with $\mathcal{O}(1)$ weights
A spectral convolution layer (g, h) is such that its connectivity matrix is of the form:

$$W_g = U^\mathsf{T} \text{diag}(\omega) U$$

where ω is obtained from smooth interpolation between a weight vector θ of size $r = \mathcal{O}(1)$ and a smoothing kernel $K \in \mathbb{R}^{n \times r}$ ie $\omega = K\theta$.

In particular, Bruna et al., 2013, argue that the second issue is answered by the fact that "in order to learn a layer in which features will be not only shared across locations but also well localized in the original domain, one can learn spectral multipliers which are smooth". An argument that was also taken up by Henaff et al., 2015. This argument is suggested by similarity with the classical Fourier transform for which spatial decay relates with smoothness in the spectral domain.

Claim 88. Smooth multipliers in the graph spectral domain produce a spatially localized operator in the vertex domain.

Although Claim 88 is true for a grid of infinite size (corresponding to the case of the discrete Fourier transform (DFT)), it is not necessarily true for a general graph in finite settings. More precisely, as mentioned by Henaff et al., 2015, this argument in classical Fourier analysis is grounded by the following expression:

$$\left|\frac{\partial^k \hat{x}}{\partial \xi^k}\right| \leq C \int |u|^k |x(u)| du \tag{25}$$

On infinite domains, the integrability of the left hand of (25) requires the summing integral to be well defined, and thus imposes x to be compactly supported, or every $|u|^k$ to be matched by the spatial decrease of $|x(u)|$ when u diverges.
However this argument doesn't need to hold on finite domains as finite sums are always defined, so it is not clear whether 88 is true for any graph.

Proposition 89. If G is a complete graph, then Claim 88 is false.

Proof. Consider a well chosen construction of $\mathcal{U}$. □

Define spatial decay, try to prove for grids
Thinking ...
Consider two graphs and their laplacians L_1 and L_2. They define the same spectral conv, iff, $L_1 L_2 = L_2 L_1$ iff $L_3 = L_1 L_2$ is a symmetric matrix *i.e.* a

laplacian of another graph.

In fact it's learned smoothness
Equivariance to L
TODO:

Polynomial spectral techniques
TODO: Chebnet Cayleynet

3.2.2 Vertex domain techniques

3.2.3 Others

3.3 Weight sharing

3.3.1 Proposed convolution operator

3.3.1.1 Definitions

Let S be a set of points, such that each one defines a set of neighbourhoods. An *entry* e of a *dataset* $\mathcal{D}$ is a column vector that can be of any size, for which each dimension represents a *value taken by* e at a certain point $u \in S$. u is said to be *activated* by e. A point u can be associated to at max one dimension of e. If it is the ith dimension of e, then we denote the value taken by e at u by either e_u or e_i. $\mathcal{D}$ is said to be *embedded* in S.
We say that two entries e and e' are *homogeneous* if they have the same size and if their dimensions are always associated to the same points.

3.3.1.2 Formal description of the *generalized* convolution

Let's denote by $\mathcal{C}$ a generalized convolution operator.We want it to observe the following conditions:

- Linearity
- Locality
- Kernel weight sharing

As $\mathcal{C}$ must be linear, then for any entry e, there is a matrix W^e such that $\mathcal{C}(e) = W^e e$. Note that unless the entries were all homogeneous, W^e depends on e.
In order to meet the locality condition, we first want that the coordinates of $\mathcal{C}(e)$ have a local meaning. To this end, we impose that $\mathcal{C}(e)$ lives in the same space than e, and that $\mathcal{C}(e)$ and e are homogeneous. Secondly, for each u, we want $\mathcal{C}(e)_u$ to be only function of values taken by e at points contained in a certain neighbourhood of u. It results that lines of W^e are generally sparse.

Let's attribute to $\mathcal{C}$ a kernel of n weights in the form of a row vector $w = (w_1, w_2, .., w_n)$, and let's define the set of allocation matrices $\mathcal{A}$ as the set of binary matrices that have at most one non-zero coordinate per column. As $\mathcal{C}$ must share its weights across the activated points, then for each row W_i^e of W^e, there is an allocation matrix $A_i^e \in \mathcal{A}$ such that $W_i^e = wA_i^e$. To maintain locality, the jth column of A_i^e must have a non-zero coordinate if and only if the ith and jth activated points are in a same neighbourhood.
Let's A^e denotes the block column vector that has the matrices A_i^e for attributes, and let's $\otimes$ denotes the tensor product. Hence, $\mathcal{C}$ is defined by a weight kernel w and an allocation map $e \mapsto A^e$ that maintains locality, such that

$$\mathcal{C}(e) = (w \otimes A^e) \ . \ e \tag{26}$$

3.3.1.3 Generalized convolution supported by an underlying graph

As vertices, the set of activated points of an entry e defines a complete oriented graph that we call G^e. If all the entries are homogeneous, then G^e is said to be *static*. In this case, we note G.
Suppose we are given $u \mapsto \mathcal{V}_u$ which maps each $u \in S$ to a neighbourhood $\mathcal{N}_u$. We then define $G^e_{\mathcal{N}}$ as the subgraph of G^e such that it contains the edge (u', u) if and only if $u' \in \mathcal{N}_u$. Let $a_{\mathcal{N}} : e \mapsto A^e$ be an allocation map such that the jth column of A_i^e have a non-zero coordinate if and only if (e_i, e_j) is an edge of $G^e_{\mathcal{N}}$.
Then the generalized convolution of e by the couple $(w, a_{\mathcal{N}})$ is supported by the underlying graph $G^e_{\mathcal{N}}$ in the sense that $W^e_{\mathcal{N}} = w \otimes a_{\mathcal{N}}(e)$ is its adjacency matrix. Note that the underlying graph of a regular convolution is a lattice and its adjacency matrix is a Toëplitz matrix.
Also note that the family $(\mathcal{V}_u)_{u \in S}$ can be seen as local receptive fields for the generalized convolution, and that the map $a_{\mathcal{N}}$ can be seen as if it were distributing the kernel weights into each $\mathcal{V}_u$.

Remarks

The generalized convolution has been defined here in view of being applied to the CNN paradigm, i.e. as an operation between a kernel weight and a vector. In the context of graph signal processing, if $G = (E, V)$ denotes a graph, then such operator $*$ with respect to the 3D tensor A between two graph signals $f, g \in \mathbb{R}^E$ would have been defined as:

$$f * g = (f^\top \otimes A) \ . \ g \tag{27}$$

$$\forall v \in V, \ (f * g)(v) = f^\top A_v \ g, \ \textit{with } A_v \in \mathcal{A} \tag{28}$$

Note that in our context, the underlying graph depends on the entry. So that if the generalized convolution is placed inside a deep neural network architecture (see section 3.3.2), then the learnt filter w will be reusable regardless of the entry's underlying structure.

3.3.1.4 Example of a generalized convolution shaped by a rectangular window

Let $\mathbb{E}$ be a two-dimensional Euclidean space and let's suppose here that $S = \mathbb{E}$.

Let's denote by $\mathcal{C}_\mathcal{R}$, a generalized convolution shaped by a rectangular window $\mathcal{R}$. We suppose that its weight kernel w is of size $N_w = (2p+1)(2q+1)$ and that $\mathcal{R}$ is of width $(2p+1)\mu$ and height $(2q+1)\mu$, where μ is a given unit scale.

Let $G^e_{p,q}$ be the subgraph of G^e such that it contains the edge (u', u) if and only if $|u'_x - u_x| \leq (p + \frac{1}{2})\mu$ and $|u'_y - u_y| \leq (q + \frac{1}{2})\mu$. In other terms, $G^e_{p,q}$ connects a vertex u to every vertex contained in $\mathcal{R}$ when centered on u.

Then, we define $\mathcal{C}_\mathcal{R}$ as being supported by $G^e_{p,q}$. As such, its adjacency

matrix acts as the convolution operator. At this point, we still need to affect the kernel weights to its non-zero coordinates, via the edges of $G^e_{p,q}$. This amounts for defining explicitely the map $e \mapsto A^e$ for $\mathcal{C}_\mathcal{R}$. To this end, let's consider the grid of same size as that rectangle, which breaks it down into $(2p+1)(2q+1)$ squares of side length μ, and let's associate a different weight to each square. Then, for each edge (u', u), we affect to it the weight associated with the square within which u' falls when the grid is centered on u. This procedure allows for the weights to be shared across the edges. It is illustrated on figure 3.1.

Note that if the entries are homogeneous and the activated points are vertices of a regular grid, then the matrix W, independant of e, is a Toëpliz matrix which acts as a regular convolution operator on the entries e. In this case, $\mathcal{C}_\mathcal{R}$ is just a regular convolution. For example, this is the case in section 3.3.1.5.

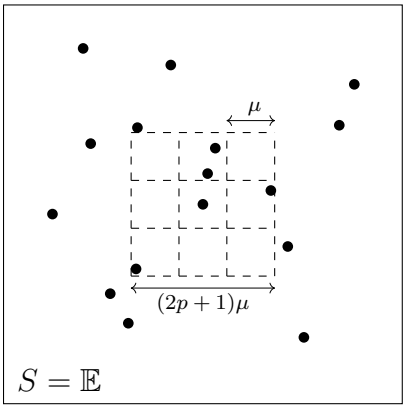

Figure 3.1: Example of a moving grid. The grid defines the allocation of the kernel weights.

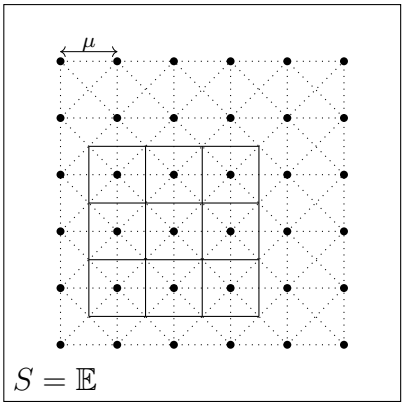

Figure 3.2: On regular domains, the moving grid allocates the weights similarly to the moving window of a standard convolution.

3.3.1.5 Link with the standard convolution on image datasets

Let $\mathcal{D}$ be an image dataset. Entries of $\mathcal{D}$ are homogeneous and their dimensions represent the value at each pixel. In this case, we can set $S = \mathbb{E}$, of

dimension 2, such that each pixel is located at entire coordinates. More precisely, if the images are of width n and height m, then the pixels are located at coordinates $(i, j) \in \{0, 1, 2, ..., n\}x\{0, 1, 2, ..., m\}$. Hence, the pixels lie on a regular grid and thus are spaced out by a constant distance $\mu = 1$.

Let's consider the static underlying graph $G_{p,q}$ and the generalized convolution by a rectangular window $\mathcal{C}_{\mathcal{R}}$, as defined in the former section. Then, applying the same weight allocation strategy will lead to affect every weight of the kernel into the moving window $\mathcal{R}$. Except on the border, one and only one pixel will fall into each square of the moving grid at each position, as depicted in figure 3.2. Indeed, $\mathcal{R}$ behaves exactly like a moving window of the standard convolution, except that it considers that the images are padded with zeroes on the borders.

3.3.2 Application to CNNs

3.3.2.1 Neural network interpretation

Let $\mathcal{L}^d$ and $\mathcal{L}^{d+1}$ be two layers of neurons, such that forward-propagation is defined from $\mathcal{L}^d$ to $\mathcal{L}^{d+1}$. Let's define such layers as a set of neurons being located in S. These layers must contain as many neurons as points that can be activated. In other terms, $S \cong \mathcal{L}^d \cong \mathcal{L}^{d+1}$. As such, we will abusively use the term of *neuron* instead of *point*.

The *generalized* convolution between these two layers can be interpreted as follow. An entry e activates the same N neurons in each layer. Then, a convolution shape takes N positions onto $\mathcal{L}^d$, each position being associated to one of the activated neurons of $\mathcal{L}^{d+1}$. At each position, connections are drawn from the activated neurons located inside the convolution shape in destination to the associated neuron. And a subset of weights from w are affected to these connections, according to a weight sharing strategy defined by an allocation map. Figure 3.3 illustrates a convolution shaped by a rectangular window.

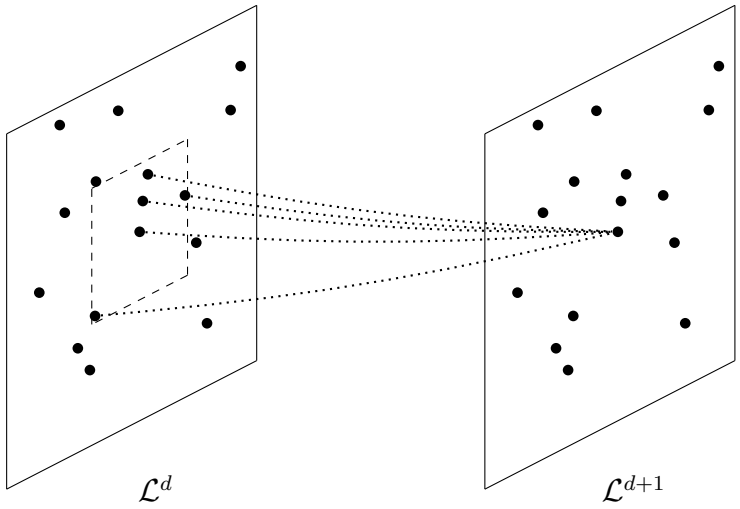

Figure 3.3: Generalized convolution between two layers of neurons.

The forward and backward propagation between $\mathcal{L}^d$ and $\mathcal{L}^{d+1}$ are applied using the described neurons and connections. After a generalized convolution operation, an activation function is applied on the output neurons. Then a pooling operation is done spatially: the input layer is divided into patches of same size, and all activated neurons included in this patch are pooled together. Unlike a standard pooling operation, the number of activated neurons in a patch may vary.

Indeed, generalized convolution layers can be vectorized. They can have multiple input channels and multiple feature maps. They shall naturally be placed into the same kind of deep neural network structure than in a CNN. Thus, they are for the irregular input spaces what the standard convolution layers are for regular input spaces.

3.3.2.2 Implementation

There are two main strategies to implement the propagations. The first one is to start from (1), derive it and vectorize it. It implies handling semi-sparse representations to minimize memory consumption and to use adequate semi-

sparse tensor products.

Instead, we decide to use the neural network interpretation and the underlying graph structure whose edges amount for neurons connections. By this mean, the sparse part of the computations is included via the use of this graph. Also, computations on each edge can be parallelized.

3.3.2.3 Forward and back propagation formulae

Let's first recall the propagation formulae from a neural network point of view.

Let's denote by e_u the value of a neuron of $\mathcal{L}^d$ located at $u \in S$, by f_v for a neuron of $\mathcal{L}^{d+1}$, and by g_v if this neuron is activated by the activation function σ.

We denote by $prev(v)$ the set neurons from the previous layer connected to v, and by $next(u)$ those of the next layer connected to u. w_{uv} is the weight affected to the connection between neurons u and v. b is the bias term associated to $\mathcal{L}^{d+1}$.

After the forward propagation, values of neurons of $\mathcal{L}^{d+1}$ are determined by those of $\mathcal{L}^d$:

$$f_v = \sum_{u \in prev(v)} e_u w_{uv} \tag{29}$$

$$g_v = \sigma(f_v + b) \tag{30}$$

Thanks to the chain rule, we can express derivatives of a layer with those of the next layer:

$$\delta_v = \frac{\partial E}{\partial f_v} = \frac{\partial E}{\partial g_v}\frac{\partial g_v}{\partial f_v} = \frac{\partial E}{\partial g_v}\sigma'(f_v + b) \tag{31}$$

$$\frac{\partial E}{\partial e_u} = \sum_{v \in next(u)} \frac{\partial E}{\partial f_v}\frac{\partial f_v}{\partial e_u} = \sum_{v \in next(u)} \delta_v w_{uv} \tag{32}$$

We call $edges(w)$ the set of edges to which the weight w is affected. If $\omega \in edges(w)$, f_{ω^+} denotes the value of the destination neuron, and e_{ω^-} denotes the value of the origin neuron.

The back propagation allows to express the derivative of any weight w:

$$\frac{\partial E}{\partial w} = \sum_{\omega \in edges(w)} \frac{\partial E}{\partial f_{\omega^+}} \frac{\partial f_{\omega^+}}{\partial w} = \sum_{\omega \in edges(w)} \delta_{\omega^+} e_{\omega^-} \tag{33}$$

$$\frac{\partial E}{\partial b} = \sum_{v} \frac{\partial E}{\partial g_v} \frac{\partial g_v}{\partial b} = \sum_{v} \delta_v \tag{34}$$

The sets $prev(v)$, $next(u)$ and $edges(w)$ are determined by the graph structure, which in turn is determined beforehand by a procedure like the one described in section 3.3.1.4.

3.3.2.4 Vectorization

Computations are done per batch of entries B. Hence, the graph structure used for the computations must contain the weighted edges of all entries $e \in B$. If necessary, entries of B are made homogeneous: if a neuron u is not activated by an entry e but is activated by another entry of B, then e_u is defined and is set to zero.

The 3D tensor counterparts of e, f, and g are thus denoted by $\mathcal{E}$, $\mathcal{F}$ and $\mathcal{G}$. Their third dimension indexes the channels (input channels or feature maps). Their submatrix along the neuron located at $x \in S$ are denoted $\mathcal{E}_x$, $\mathcal{F}_x$ and $\mathcal{G}_x$, rows are indexing entries and columns are indexing channels. The counterparts of w and b are $\mathcal{W}$ and β. The first being a 3D tensor and the second being a vector with one value per feature map. $\widetilde{\beta}$ denotes the 3D tensor obtained by broadcasting β along the two other dimensions of $\mathcal{F}$. $\mathcal{W}_w$ denotes the submatrix of $\mathcal{W}$ along the kernel weight w. Its rows index the feature maps and its columns index the input channels.

With these notations, the vectorized counterparts of the formulae from sec-

tion 3.3.2.3 can be obtained in the same way:

$$\mathcal{F}_v = \sum_{u \in prev(v)} \mathcal{E}_u \mathcal{W}_{w_{uv}}^{\top} \tag{35}$$

$$\mathcal{G} = \sigma(\mathcal{F} + \widetilde{\beta}) \tag{36}$$

$$\Delta = \left(\frac{\partial E}{\partial \mathcal{F}}\right) = \left(\frac{\partial E}{\partial \mathcal{G}}\right) \circ \ \sigma^{'}(\mathcal{F} + \widetilde{\beta}) \tag{37}$$

$$\left(\frac{\partial E}{\partial \mathcal{E}}\right)_u = \sum_{v \in next(u)} \Delta_v \mathcal{W}_{uv} \tag{38}$$

$$\left(\frac{\partial E}{\partial \mathcal{W}}\right)_w = \sum_{\omega \in edges(w)} \Delta_{\omega^+}^{\top} \mathcal{E}_{\omega^-} \tag{39}$$

$$\frac{\partial E}{\partial \beta} = \sum_j \left(\sum_v \Delta_v\right)_{j\text{-}th\ column} \tag{40}$$

Where $\circ$ and $^{\top}$ respectively denotes Hadamard product and transpose.

3.3.3 Experiments

In order to measure the gain of performance allowed by the generalized CNN over MLP on irregular domains, we made a series of benchmarks on distorded versions of the MNIST dataset LeCun et al., 1998b, consisting of images of 28x28 pixels. To distort the input domain, we plunged the images into a 2-d euclidean space by giving entire coordinates to pixels. Then, we applied a gaussian displacement on each pixel, thus making the data irregular and unsuitable for regular convolutions. For multiple values of standard deviation of the displacement, we trained a generalized CNN and compared it with a MLP that has the same number of parameters. We choosed a simple yet standard architecture in order to better see the impact of generalized layers. The architecture used is the following: a generalized convolution layer with relu Glorot et al., 2011 and max pooling, made of 20 feature maps, followed by a dense layer and a softmax output layer. The generalized convolution

is shaped by a rectangular window of width and height 5μ where the unit scale μ is chosen to be equal to original distance between two pixels. The max-pooling is done with square patches of side length 2μ. The dense layer is composed of 500 hidden units and is terminated by relu activation as well. In order to have the same number of parameters, the compared MLP have 2 dense layers of 500 hidden units each and is followed by the same output layer. For training, we used stochastic gradient descent Bottou, 2010 with Nesterov momentum Sutskever et al., 2013 and a bit of L2 regularization Ng, 2004.

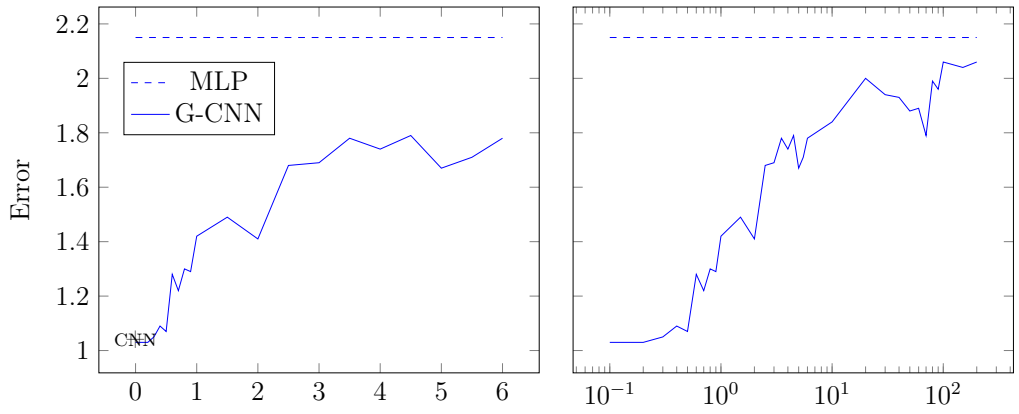

Figure 3.4: Results for Generalized CNN and MLP, both with 500 parameters and 15 epochs.

The plot drawn on figure 3.4 illustrates the gain of performance of a generalized convolutional layer over a dense layer with equal number of parameters. After 15 epochs for both, it shows that the generalized CNN on a distorded domain performs better than the MLP. Indeed, in the case of no domain distortion, the score is the same than a CNN with zero-padding. The error rate goes up a bit with distortion. But even at 200μ, the generalized CNN still performs better than the MLP.

3.3.4 Proposed Method

We propose to introduce another type of layer, that we call *receptive graph layer*. It is based on an adjacency matrix and aims at extending the principle of convolutional layers to any domain that can be described using a graph.
Consider an adjacency matrix A that is well fitted to the signals to be learned, in the sense that it describes an underlying graph structure between the input features. We define the receptive graph layer associated with A using the product between a third rank tensor S and a weight kernel W. For now, the tensor W would be one-rank containing the weights of the layer and S is of shape $n \times n \times \omega$, where $n \times n$ is the shape of the adjacency matrix and ω is the shape of W.
On the first two ranks, the support of S must not exceed that of A, such that $A_{ij} = 0 \Rightarrow \forall k, S_{ijk} = 0$.
Overall, we obtain:

$$\mathbf{y} = f(W \cdot S \cdot \mathbf{x} + \mathbf{b}) \ ,$$

where here $\cdot$ denotes the tensor product.
Intuitively, the values of the weight kernel W are linearly distributed to pairs of neighbours in A with respect to the values of S. For this reason, we call S the *scheme* (or *weight sharing scheme*) of the receptive graph. In a sense, this scheme tensor is to the receptive graph what the adjacency matrix is to the graph. An example is depicted in Figure 3.5.
Alike convolution on images, W is extended as a third-rank tensor to include multiple input and output channels (also known as feature maps). It is worth mentioning that an implementation must be memory efficient to take care of a possibly large sparse S.

3.3.5 Training

The proposed formulation allows to learn both S and W. We perform the two jointly. Learning W amounts to learning weights as in regular CNNs,

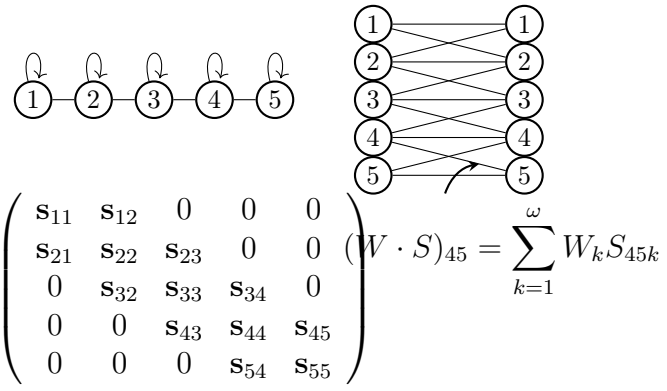

Figure 3.5: Depiction of a graph, the corresponding receptive graph of the propagation and its associated weight sharing scheme S. Note that $\mathbf{s}_{ij} = S_{ij\cdot}$ are vector slices of S along the first two ranks, $\mathbf{s}_{ij}$ determines how much of each weight in W is allocated for the edge linking vertex i to vertex j.

whereas learning S amounts to learning how these weights are tied over the receptive fields. We also experiment a fine-tuning step, which consists in freezing S in the last epochs. Indeed, when a weight sharing scheme can be decided directly from the underlying structure, it is not necessary to train S.

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 third dimension. Therefore, the vectors on the third rank of S can be interpreted as performing a weighted average of the parameters in W.

We test two types of initialization for S. The first one consists in distributing one-hot-bit vectors along the third rank. We impose that for each receptive field, a particular one-hot-bit vector can only be distributed at most once more than any other. We refer to it as one-hot-bit initialization. The second one consists in using a uniform random distribution with limits as described in Glorot and Bengio, 2010.

3.3.6 Genericity

For simplicity we restricted our explanation to square adjacency matrices. In the case of oriented graphs, one could remove the rows and columns of zeros and obtain a receptive graph with a distinct number of neurons in the input (n) than in the output (m). As a result, receptive graph layers extend usual ones, as explained here:

1. To obtain a fully connected layer, one can choose ω to be of size nm and S the matrix of vectors that contains all possible one-hot-bit vectors.

2. To obtain a convolutional layer, one can choose ω to be the size of the kernel. S would be one-hot-bit encoded along its third rank and circulant along the first two ranks. A stride > 1 can be obtained by removing the corresponding rows.

3. Similarly, most of the layers presented in related works can be obtained for an appropriate definition of S.

In our case, S is more similar to that obtained when considering convolutional layers, with the noticeable differences that we do not force which weight to allocate for which neighbor along its third rank and it is not necessarily circulant along the first two ranks.

3.3.7 Discussion

Although we train S and W, the layer propagation is ultimately handled by their tensor product. That is, its output is determined by $\Theta \cdot \mathbf{x}$ where $\Theta = S \cdot W$. For the weight sharing to make sense, we must then not over-parameterize S and W over Θ. If we call l the number of non-zeros in A and $w \times p \times q$ the shape of W, then the former assumption requires $lw + wpq \leq lpq$ or equivalently $\frac{1}{w} \geq \frac{1}{pq} + \frac{1}{l}$. It implies that the number of weights per filter w must be lower than the total number of filters pq and than the number of edges l.

Note that without the constraint that the support of S must not exceed that of A (or if the used graph is complete), the proposed formulation could also be applied to structure learning of the input features space Richardson, 1996; Kwok and Yeung, 1997. That is, operations on S along the third rank might be exploitable in some way, e.g. dropping connections during training Han et al., 2015 or discovering some sort of structural correlations. However, even if this can be done for toy image datasets, such S wouldn't be sparse and would lead to memory issues in higher dimensions. So we didn't include these avenues in the scope of this paper.

3.3.8 Experiments

Description

We are interested in comparing various receptive graph layers with convolutional ones. For this purpose, we use image datasets, but restrain priors about the underlying structure.

We first present experiments on MNIST LeCun et al., 1998b. It contains 10 classes of gray levels images (28x28 pixels) with 60'000 examples for training, 10'000 for testing. We also do experiments on a scrambled version to hide the underlying structure, as done in previous work Chen et al., 2014. Then we present experiments on Cifar10 Krizhevsky, 2009. It contains 10 classes of RGB images (32x32 pixels) with 50'000 examples for training, 10'000 for testing.

Because receptive graph layers are wider than their convolutional counterparts (lw more parameters from S), experiments are done on shallow (but wide) networks for this introductory paper. Also note that they require $w+1$ times more multiply operations than a convolution lowered to a matrix multiplication Chetlur et al., 2014. In practice, they roughly took 2 to 2.5 more time.

Experiments with grid graphs on MNIST

Here we use models composed of a single receptive graph (or convolutional) layer made of 50 feature maps, without pooling, followed by a fully connected layer of 300 neurons, and terminated by a softmax layer of 10 neurons. Rectified Linear Units Glorot et al., 2011 are used for the activations and a dropout Srivastava et al., 2014 of 0.5 is applied on the fully-connected layer. 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.

We consider a grid graph that connects each pixel to itself and its 4 nearest neighbors (or less on the borders). We also use 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). Here we use one-hot-bit initialization. We test the model under two setups: either the ordering of the node is unknown, and then the one-hot-bit vectors are distributed randomly and modified upon training ; either an ordering of the node is known, and then the one-hot-bit vectors are distributed in a circulant fashion in the third rank of S which is freezed in this state. We use the number of nearest neighbors as for the dimension of the third 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. Table 3.1 summarizes the obtained results. The ordering is unknown for the first result given, and known for the second result between parenthesis.

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

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 3.1: Error rates on powers of the grid graphs on MNIST.

In Figure 3.6, 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. We observe that they have little influence on the performance and sometimes improve it a bit. Thus, we use them as optional hyperparameters.

Experiments with covariance graphs on Scrambled MNIST

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 third rank of S is chosen equal to k and its weights are initialized random uniformly Glorot and Bengio, 2010. The receptive graph layers 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 3.2. We observe that the receptive graph layers outperforms 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 thanks to its

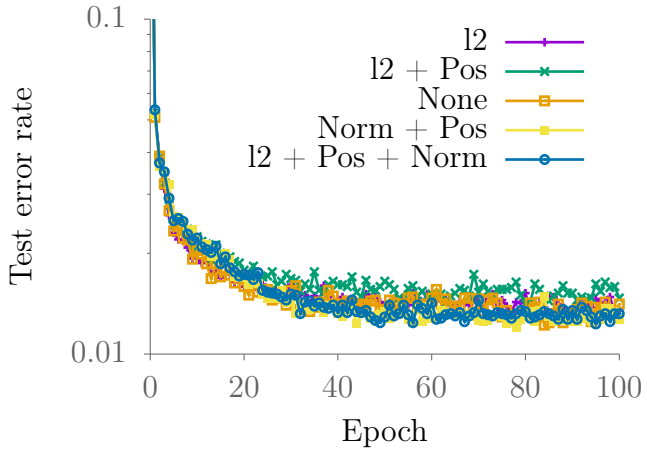

Figure 3.6: Evolution of the test error rate when learning MNIST using the square of a grid graph and for various normalizations, as a function of the epoch of training. The legend reads: "l2" means ℓ_2 normalization of weights is used (with weights 10^{-5}), "Pos" means parameters in S are forced to being positive, and "Norm" means that the ℓ_1 norm of each vector in the third dimension of S is forced to 1.

graph.

Experiments with shallow architectures on Cifar10

On Cifar10, we made experiments on shallow CNN architectures and replaced convolutions by receptive graphs. The first architecture used is the same than in the previous experiments on MNIST, and the second one is a variant of AlexNet Krizhevsky et al., 2012 using little distortion on the input that we borrowed from a tutorial of tensorflow Abadi et al., 2015. The latter is composed of two 5x5 convolutional layers of 64 feature maps, with max pooling and local response normalization, followed by two fully connected layers of 384 and 192 neurons. 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

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

Table 3.2: Error rates when topology is unknown on scrambled MNIST.

stochastic gradient descent on 375 epochs where S is freezed on the 125 last ones. Circulant one-hot-bit intialization is used. These are weak classifiers for Cifar10 but they are enough to analyse the usefulness of the proposed layer. Exploring deeper architectures is left for further work. Results are summarized in table 3.3. "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	# convs	Learning S	None	Pos	Norm	Both
Conv5x5	1	No	/	/	/	66.1%
Grid^2	1	Yes	67.3%	66.8%	67.1%	67.0%
Conv5x5	2	No	/	/	/	87.0%
Conv5x5	2	Yes	87.3%	86.9%	86.9%	87.5%
Grid^2	2	Yes	87.1%	87.3%	87.6%	87.5%

Table 3.3: Accuracies of shallow networks on CIFAR10.

The receptive graph layers are able to outperform the corresponding CNNs by a small amount in the tested configurations, opening the way for more complex architectures.

Chapter 4

Industrial applications

TODO:

Conclusion

TODO:

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. 27, 91).

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. 19).

Bass, Jean (1968). "Cours de mathématiques". In: (cit. on p. 9).

Bengio, Yoshua (2009). "Learning deep architectures for AI". In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on p. 18).

Bianchini, Monica and Franco Scarselli (2014). "On the complexity of neural network classifiers: A comparison between shallow and deep architec-

tures". In: *IEEE transactions on neural networks and learning systems* 25.8, pp. 1553–1565 (cit. on p. 19).

Bottou, Léon (2010). "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, pp. 177–186 (cit. on pp. 31, 84).

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 pp. 71–73).

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. 59).

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. 88).

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. 88).

Chung, Fan R. K. (1996). *Spectral Graph Theory (CBMS Regional Conference Series in Mathematics, No. 92)*. American Mathematical Society. ISBN: 0821803158 (cit. on p. 71).

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. 17).

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. 19).

Cohen, Nadav, Ronen Tamari, and Amnon Shashua (2018a). "Boosting Dilated Convolutional Networks with Mixed Tensor Decompositions". In:

International Conference on Learning Representations. URL: https://openreview.net/forum?id=S1JHhv6TW (cit. on pp. 18, 20).

Cohen, Taco and Max Welling (2016). "Group equivariant convolutional networks". In: *International Conference on Machine Learning*, pp. 2990–2999 (cit. on p. 116).

Cohen, Taco S., Mario Geiger, Jonas Köhler, and Max Welling (2018b). "Spherical CNNs". In: *International Conference on Learning Representations*. URL: https://openreview.net/forum?id=Hkbd5xZRb (cit. on p. 116).

Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4, pp. 303–314 (cit. on p. 17).

Delalleau, Olivier and Yoshua Bengio (2011). "Shallow vs. deep sum-product networks". In: *Advances in Neural Information Processing Systems*, pp. 666–674 (cit. on pp. 18, 19).

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. 18, 20).

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. 19).

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. 18, 86, 90).

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. 17, 18, 83, 89).

Hackbusch, Wolfgang (2012). *Tensor spaces and numerical tensor calculus*. Vol. 42. Springer Science & Business Media (cit. on pp. 9, 10).

Han, Song, Jeff Pool, John Tran, and William Dally (2015). "Learning both weights and connections for efficient neural network". In: *Advances in Neural Information Processing Systems*, pp. 1135–1143 (cit. on p. 88).

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. 18).

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. 17).

— (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778 (cit. on pp. 16, 20).

Henaff, Mikael, Joan Bruna, and Yann LeCun (2015). "Deep convolutional networks on graph-structured data". In: *arXiv preprint arXiv:1506.05163* (cit. on pp. 71, 73).

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. 16).

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. 18).

Hoogeboom, Emiel, Jorn W.T. Peters, Taco S. Cohen, and Max Welling (2018). "HexaConv". In: *International Conference on Learning Representations*. URL: `https://openreview.net/forum?id=r1vuQG-CW` (cit. on p. 116).

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2, pp. 251–257 (cit. on p. 17).

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. 17, 30, 31).

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. 16, 20).

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. 17).

Kingma, Diederik and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (cit. on p. 89).

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. 17).

Krizhevsky, Alex (2009). "Learning multiple layers of features from tiny images". In: (cit. on p. 88).

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. 16, 18, 29, 91).

Kwok, Tin-Yau and Dit-Yan Yeung (1997). "Constructive algorithms for structure learning in feedforward neural networks for regression prob-

lems". In: *IEEE Transactions on Neural Networks* 8.3, pp. 630–645 (cit. on p. 88).

LeCun, Y. (1987). "Modeles connexionnistes de l'apprentissage (connectionist learning models)". PhD thesis. Université P. et M. Curie (Paris 6) (cit. on p. 21).

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. 16, 29).

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. 16, 17).

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998a). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324 (cit. on p. 30).

LeCun, Yann, Corinna Cortes, and Christopher JC Burges (1998b). *The MNIST database of handwritten digits* (cit. on pp. 83, 88).

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. 19).

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. 17).

Marcus, Marvin (1975). "Finite dimensional multilinear algebra". In: (cit. on p. 9).

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. 19).

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. 19).

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 pp. 84, 89).

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. 18).

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. 21).

Pan, Xingyuan and Vivek Srikumar (2016). "Expressiveness of rectifier networks". In: *International Conference on Machine Learning*, pp. 2427–2435 (cit. on p. 19).

Pascanu, Razvan, Guido Montufar, and Yoshua Bengio (2013). "On the number of response regions of deep feed forward networks with piece-wise linear activations". In: *arXiv preprint arXiv:1312.6098* (cit. on pp. 18, 19).

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. 19).

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. 19).

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. 19).

Richardson, Thomas (1996). "A discovery algorithm for directed cyclic graphs". In: *Proceedings of the Twelfth international conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., pp. 454–461 (cit. on p. 88).

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. 16, 22, 23).

Schwartz, Laurent (1957). *Théorie des distributions*. Vol. 2. Hermann Paris (cit. on p. 43).

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. 71).

Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (cit. on p. 16).

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 pp. 30, 89).

Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton (2013). "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th international conference on machine learning (ICML-13)*, pp. 1139–1147 (cit. on p. 84).

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. 16, 20).

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. 20).

Weinstein, Alan (1996). "Groupoids: unifying internal and external symmetry". In: *Notices of the AMS* 43.7, pp. 744–752 (cit. on p. 116).

Widrow, Bernard and Marcian E Hoff (1960). *Adaptive switching circuits*. Tech. rep. STANFORD UNIV CA STANFORD ELECTRONICS LABS (cit. on pp. 22, 23).

Wikipedia, contributors (2018a). *Cayley graph — Wikipedia, The Free Encyclopedia*. [Online; accessed April-2018]. URL: `https://en.wikipedia.org/wiki/Cayley_graph` (cit. on p. 59).

— (2018b). *Convolution theorem — Wikipedia, The Free Encyclopedia*. [Online; accessed April-2018]. URL: `https://en.wikipedia.org/wiki/Convolution_theorem` (cit. on p. 71).

— (2018c). *Feedforward neural network — Wikipedia, The Free Encyclopedia*. [Online; accessed April-2018]. URL: `https://en.wikipedia.org/wiki/Feedforward_neural_network` (cit. on p. 16).

Williamson, S Gill (2015). "Tensor spaces-the basics". In: *arXiv preprint arXiv:1510.02428* (cit. on p. 9).

Zell, Andreas (1994). *Simulation neuronaler netze*. Vol. 1. Addison-Wesley Bonn (cit. on p. 16).

Zoph, Barret and Quoc V Le (2016). "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (cit. on p. 16).

Chapter 0

Trash bin and more drafts

... that we may use in some section.

TODO: Rework 1.1

0.1 Naming conventions

0.1.1 Basic notions

Let's recall the naming conventions of basic notions.
A *function* $f : E \to F$ maps objects $x \in E$ to objects $y \in F$, as $y = f(x)$.
Its *definition domain* $\mathcal{D}_f = E$ is the set of objects onto which it is defined.
We will often just use the term *domain*.
We also say that f is *taking values* in its *codomain* F.
The *image per* f of the subset $U \subset E$, denoted $f(U)$, is $\{y \in F, \exists x \in E, y = f(x)\}$.
The *image of* f is the image of its domain. We denote $\mathcal{I}_f$.
A vector space E, which we will always assume to be finite-dimensional in our context, is defined as $\mathbb{R}^n$, and is equipped with pointwise addition and scalar multiplication.
A *signal* s is a function taking values in a vector space. In other words, a signal can also be seen as a *vector* with an *underlying structure*, where the vector is composed from its image, and the underlying structure is defined by its *domain*.

For example, images are signals defined on a set of pixels. Typically, an image s in RGB representation is a mapping from pixels p to a 3d vector space, as $s_p = (r, g, b)$.
TODO?: figure

0.1.2 Graphs and graph signals

TODO: more defs on grid graphs and other graphs

A *graph* $G = (V, E)$ is defined as a set of nodes V, and a set of edges $E \subseteq \binom{V}{2}$. The words *node* and *vertex* will be used equivalently, but we will rather use the first.

A *graph signal*, or *graph-structured signal* is a signal defined on the nodes of a graph, for which the underlying structure is the graph itself. A *node signal* is a signal defined on a node, in which case it is a *node embedding* in a vector space.

Although this is rarely seen, a signal can also be defined on the edges of a graph, or on an edge. We then coin it respectively *dual graph signal*, or *edge signal / edge embedding*.

Graph-structured data can refer to any of these type of signals.

0.1.3 Data and datasets

A dataset of signals is said to be *static* if all its signals share the same underlying structure, it is said to be *non-static* otherwise.
For image datasets, being non-static would mean that the dataset contains images of different sizes or different scales. For graph signal datasets, it would mean thats the underlying graph structures of the signals are different.

The point in specifying that objects of a dataset of a machine learning task are signals is that we can hope to leverage their underlying structure.

TODO: figure

0.2 Disambiguation of the subject

This thesis is entitled *Deep learning models for data without a regular structure*. So either the data of interest in this manuscript do not have any structure, or either their structure is not regular.

0.2.1 Irregularly structured data

By structured data, we mean that there exists an underlying structure over which the data is defined. This kind of data are usually modelized as signals defined over a domain. These domains are then composed of objects that are related together by some sort of structural properties. For example, pixels of images can be seen as located on a grid with integer spatial coordinates (a 2d cartesian grid graph).
It then come in handy to define the notions of structure and regularity with the help of graph signals.

Definition 90. Structure
Let $s : D \to F$ be a signal defined over a finite domain.
An *underlying structure* of the signal s is a graph G that has the domain of s for nodes.
A dataset is said to be *structured*, if its objects can be modelized as signals with an underlying structure.
It is said to be *static* if all its objects share the same underlying structure, and *non-static* otherwise.

In other words, we chose to define "structured data" as "graph-structured data" by some graph. Hence we need to specify for which graphs this structure would be said to be regular, and for which it would not.

Definition 91. Regularity
An underlying structure is said to be *regular*, if it is a regular grid graph. It is said to be *irregular* otherwise.
A dataset is said to be *regularly structured*, if the underlying structures of its objects are regular. It is said to be *irregularly structured* otherwise.

TODO: examples

0.2.2 Unstructured data

Data can also be unstructured. If the data is not yet embedded into a finite dimensional vector space, then we will be interested in embedding techniques used in representation learning. In the other case, it is often possible to fall back to the case of irregularly structured data. For example, vectors can be seen as signals defined over the canonical basis of the vector space, and the vectors of this basis can be related together by their covariances through the dataset. It is typical to use the graph structure that has the canonical basis for nodes, with edges obtained by covariance thresholding.
TODO: examples
What follows is a draft

0.3 Datasets

0.4 Tasks

0.5 Goals

0.6 Invariance

In order to be observed, invariances must be defined relatively to an observation. Let's give a formal definition to support our discussion.
...

0.7 Methods

0.8 Expressivity analysis of dense versus sparse connectivity

Let consider a tensor input x of a neural network layer l. Without loss of generality, we consider that x is a matrix of shape $n \times p$. Its rows are supposed structured by a graph $G =< V, E >$, with $|V| = n$, its columns are its feature maps.

In what follows, we discuss the expressivity and efficiency of a dense layer with x as input versus a layer that would leverage G. We start with the regular case and continue onto non-regular structures.

0.8.1 Strong regular case

In the strong regular case, G is a lattice graph such that a convolution is defined naturally on it. For example, this is the case where rows of x defines ticks of a time series, or flattened pixels of an image.

Let consider a convolutional layer $c = (g_c, h_c)$ with padding, defined by q filters of width k. Define y_c its output of shape $n \times q$.

We are interested in knowing if there exists a dense layer that can efficiently replicate c.

Its connectivity matrix W_c is of shape $npxnq$. Obviously, the function g_c can be replicated by a dummy dense layer $d = (g_c, h_d)$ through W_c. However, whereas c has only kpq weights, d has n^2pq. If we consider the families of neural networks $\mathcal{C}$, $\mathcal{D}$ spaned by their weights θ_c, θ_d, then we realize the $\mathcal{C}$ is less expressive, but in the same time it is more efficient at representing its functions.

Let's define the notion of partial expressivity with respect to a family of functions.

Let $\mathcal{F}$ a family of functions, $\mathcal{L}$ a family of layer functions, and ϵ the approximation coefficient. For $f \in \mathcal{F}$, define $S_\epsilon(\mathcal{L}, f) = \{l \in \mathcal{L}, d(l, f) < \epsilon\}$ and

$S_\epsilon(\mathcal{L}, \mathcal{F}) = \bigcup_{f \in \mathcal{F}} S_\epsilon(\mathcal{L}, f)$.
TODO: reword above

By abusing and anticipating future correction of this manuscript, we consider that $\mathcal{C}$ and $\mathcal{D}$ are vector spaces. We are interesting in 1. proving that $S_\epsilon(\mathcal{C}, \mathcal{F})$ and $S_\epsilon(\mathcal{D}, \mathcal{F})$ are also vector spaces, and 2. analysing for which $\mathcal{F}$, $\frac{dim(S_\epsilon(\mathcal{C},\mathcal{F}))}{dim(S_\epsilon(\mathcal{D},\mathcal{F}))}$ is maximized.
Obviously 1. is false, so 2. is ill-posed (this draft is to be reworded afterward). Instead of using dim, we should rather use $card$. However they are potentially infinite families so we should rather use a notion of volume, except if we discretize. So let's discretize.
By the way, ''modified'' 2. is trivially maximized for $\mathcal{F} = \mathcal{C}$ (and then the ratio equals 1), so let's weaken $\mathcal{F}$ and say it's any family with translation equivariance. We are then interested in proving that if $\mathcal{F}$ is the family on translation equivariant function (on this domains that has to be specified when rewriting this section), then $\frac{card(S_\epsilon(\mathcal{C},\mathcal{F}))}{card(S_\epsilon(\mathcal{D},\mathcal{F}))}$ is close to 1. Equivariant in our context means commuting with translations (we should rather use the latter expression btw).
The result might be obtained without discretizing as convolutions with padding commutes with translations. Let's guess that they are close to other commuters. In fact that is even it. Proof with Fourier analysis.

0.8.2 Draft

The only dense layer that replicate g_c is obtained through the connectivity matrix W_c. $\mathcal{D}$ is more expressive, however less efficient as we are looking for equivariant functions. It happens that equivariant functions are exactly convolutions with padding.

0.9 Conv drafts

TODO: point

In particular, we have

$$\begin{aligned}
\forall s \in \mathcal{S}(\Gamma), \widetilde{\varphi}(s) &= \widetilde{\varphi}\left(\sum_{g\in\Gamma} s[g]\delta_g\right) \\
&= \sum_{g\in\Gamma} s[g]\widetilde{\varphi}\left(\delta_g\right) \\
&= \sum_{g\in\Gamma} s[g]\delta_{\varphi(g)} \\
&= \sum_{v\in V} s[\varphi^{-1}(v)]\delta_v \\
\widetilde{\varphi}(s) &= \sum_{v\in V} \widetilde{\varphi}(s)[v]\delta_v
\end{aligned}$$

So $\widetilde{\varphi}(s)[v] = s[\varphi^{-1}(v)]$ and $\widetilde{\varphi}(s)[\varphi(g)] = s[g]$. Let's simplify the notations with $\widetilde{\varphi}(s) = t$ and $\varphi(g) = v$, *i.e.* $t[v] = s[g]$ as expected. We then define the group convolution on $\mathcal{S}(V)$ as

$$\begin{aligned}
(t_1 * t_2)[v] &= (s_1 * s_2)[g] \\
&= \sum_{h\in\mathcal{G}} s_1[h]\, s_2[h^{-1}g] \\
&= \sum_{u\in V} s_1[\varphi^{-1}(u)]\, h_u(s_2)[\varphi^{-1}(v)] \\
&= \sum_{u\in V} t_1[u]\, \widetilde{\varphi}(h_u(s_2))[v]
\end{aligned}$$

$$(t_1 * t_2)[v] = \sum_{u \in V} t_1[u]\, h_u(t_2)[v] \tag{41}$$

$$\tag{42}$$

TODO: stop sign

Recall that

$$\begin{aligned}\delta_g[h] &= \begin{cases} 1 & \text{if } h = g \Leftrightarrow \varphi(h) = \varphi(g) \\ 0 & \text{otherwise} \end{cases} \\ &= \delta_{\varphi(g)}[\varphi(h)]\end{aligned}$$

$$s = \sum_{v \in V} s[v]\, \delta_v$$

TODO: lemme on existence of uncountable linearly independent irrational family ?

Proposition 92. The group convolution on $\mathcal{S}(\Gamma)$ has a unique neutral element which is the dirac signal on the identity tranformation.

Proof. Denote δ a neutral element for the group convolution. Note as because of the commutativity the group convolution, a left neutral element is also a right neutral element. We have

$$s[h] = (\delta * s)[h] = \sum_{g \in \Gamma} \delta[g]\, s[g^{-1}h]$$

which is true for any real valued signal. By chosing a signal π having linearly

independant irrational entries (and using the axiom of choice in case G is not finite), we obtain that

$$\delta[g] = \begin{cases} 1 \text{ if } g = \mathrm{Id} \\ 0 \text{ otherwise} \end{cases} \quad i.e. \quad \delta = \delta_{\mathrm{Id}}$$

Conversely, $(\delta_{\mathrm{Id}} * s)[h] = 1.s[\mathrm{Id}^{-1}h] = s[h]$. □

In other therms, if there is an isomorphism between Γ and V, the group structure pass to V as well as the definition of the group convolution.

To alleviate this issue, let's introduce the neutral elements δ of the convolution, and the neutral element $\mathrm{Id} \in \Phi^*(V)$.

With the help of δ, we follow the same process as in the proof of Proposition 80, see (14), to construct the class of group convolutional operators which defines exactly the class of linear transformations that are equivariant to a certain group.

On graphs, this could be used provided we defined meaningful translations beforehand (see Section **??**). Another possibilty would be to search for invariances with respect to graph equivariances and derive a convolution operator similarly than for translations. This approach, which uses group convolutions (Weinstein, 1996), has already been discussed on regular domain to extend CNNs to other invariances than translational ones (Cohen and Welling, 2016; Hoogeboom et al., 2018), as well as on spherical domain with rotation equivariant CNNs (Cohen et al., 2018b). As stated from the previous remark, the big advantage of this approach is that there is no loss of expressivity. However on graphs, this would be more challenging as it's not likely there exists transformations with equivariances. However, let's suppose we found such a set of transformations on a graph, then for Proposition 80 to hold (instead as for regular translations), we see in the proof that they need to be bijective (13) and vertex dependent 14.

0.9.1

Definition 93. Grounded set of transformations
A set of transformations over a graph $G = \langle V, E \rangle$, *grounded* on a vertex $v_0 \in V$, denoted $\mathcal{P}_{v_0} \subset \Phi(V)$, is a set that is in one-to-one correspondence with V, such that $\forall v \in V, \exists! p_v \in \mathcal{P}_{v_0}, p_v(v_0) = v$.

We have $\mathcal{P}_{v_0} = \text{order}(G) \in \mathbb{N} \cup \{+\infty\}$. For notational convenience we drop the subscript $_{v_0}$ in what follows.

Definition 94. $\mathcal{P}$-equivariant convolution operator
Let $G = \langle V, E \rangle$ a graph, not necessarily a grid. Let $\mathcal{P}$ a grounded set of transformations. Then, the $\mathcal{P}$-equivariant convolution operator f_w is defined as

$$\forall s \in \mathcal{S}(V), f_w(s) = s *_{\mathcal{P}} w = \sum_v s[v]\, p_v(w)$$

Claim 95. Characterization of $\mathcal{P}$-eq. convolution operator
The class of linear graph signal transformations that are equivariant to a grounded set $\mathcal{P}$ is exactly the class of $\mathcal{P}$-equivariant convolutive operations.

Proof. By construction of $\mathcal{P}$-equivariant convolutions, the proof is similar to the one of Proposition 80. □

Contents

Chapter 0

Keywords and temptative titles

Index terms— Deep learning, representation learning, propagation learning, visualization, structured, unstructured regular, irregular, covariant, invariant, equivariant, tensor, scheme, weight sharing, graphs, manifold, euclidean, signal processing, graph signal processing, time series, time series database, distributed application, spatial-time series, geo time series, industrial applications, warp 10, warpscript, ...

Temptative titles

- Learning propagational representations of irregular and unstructured data
- Learning representations of unstructured or irregularly structured datasets
- Propagational learning of unstructured or irregularly structured datasets
- Learning tensorial representation of irregular and unstructured data
- Tensorial representation of propagation in deep learning for irregular and unstructured dataset
- Structural representation learning for irregular or unstructured data

- Word for both "irregularly structured" + "unstructured" = ? (maybe "unorthodox" ?)
- Unorthdox deep learning
- ...
- Deep learning of unstructured or irregularly structured datasets
- Deep learning models for data without a regular structure
- On structures in deep learning
- On deep learning for when data is lacking a regular structure
- Deep learning for non regularly structured data