# Contents

**1 Presentation of the field** **3**
1.1 Tensors . . . 5
1.1.1 Definition . . . 5
1.1.2 Manipulation . . . 7
1.1.3 Binary operations . . . 9
1.2 Deep learning . . . 13
1.2.1 Neural networks . . . 13
1.2.2 Interpretation . . . 19
1.2.3 Training . . . 20
1.2.4 Historical advances . . . 23
1.2.5 Common layers . . . 26
1.3 Deep learning on graphs . . . 32
1.3.1 Graph and signals . . . 32
1.3.2 Learning tasks . . . 34
1.3.3 Spectral methods . . . 36
1.3.4 Vertex-domain methods . . . 40

**Bibliography** **43**

# Chapter 1

# Presentation of the field

## 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 $\mathbb{R}$.

## Contents

**1.1 Tensors . . . 5**
1.1.1 Definition . . . 5
1.1.2 Manipulation . . . 7
1.1.3 Binary operations . . . 9
**1.2 Deep learning . . . 13**
1.2.1 Neural networks . . . 13
1.2.2 Interpretation . . . 19
1.2.3 Training . . . 20
1.2.4 Historical advances . . . 23
1.2.5 Common layers . . . 26
**1.3 Deep learning on graphs . . . 32**
1.3.1 Graph and signals . . . 32
1.3.2 Learning tasks . . . 34
1.3.3 Spectral methods . . . 36
1.3.4 Vertex-domain methods . . . 40

## 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 $t[i_1, i_2, \ldots, i_r]$.
More precisely, if $\mathbb{T} = \bigotimes_{k=1}^{r} \mathbb{V}_k$, with bases $((e_k^i)_{i=1,\ldots,n_k})_{k=1,\ldots,r}$, then we have

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

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 $t' = t[l_1{:}u_1, l_2{:}u_2, \ldots, l_r{:}u_r]$, 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

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

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 **??**).

**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:

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

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

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

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

## Einstein summation convention

The Einstein summation convention is a notational convention to write a sum-product expression as a product expression. The summation indices are those that appear simultaneously in the superscript of the left operand and in the subscript of the right one, if subscripts precede superscripts in the notation, or else vice-versa. For example, a dot product is written $u_k v^k = \lambda$ and a matrix product is written ${A_i}^k {B_k}^j = {C_i}^j$.
The tensor contraction of Definition 9 can be rewritten using this convention:

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

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

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

$$T_{1\,g_i(i_1^{(1)},\dots,i_{r_1-s}^{(1)})}{}^{g_k(k_1,\dots,k_s)} T_{2\,g_k(k_1,\dots,k_s)}{}^{g_j(i_{s+1}^{(2)},\dots,i_{r_2}^{(2)})} = T_{3\,g_i(i_1^{(1)},\dots,i_{r_1-s}^{(1)})}{}^{g_j(i_{s+1}^{(2)},\dots,i_{r_2}^{(2)})}$$

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:

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

**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 Deep learning

### 1.2.1 Neural networks

A feed-forward neural network could originally be formalized as a composite function chaining linear and non-linear functions (Rumelhart et al., 1985; LeCun et al., 1989; LeCun, Bengio, et al., 1995). That whas still the case in 2012 when important breakthroughs regenerated a surge of interest in the field (Hinton et al., 2012; Krizhevsky et al., 2012; Simonyan and Zisserman, 2014). However, in more recent years, more complex architectures have emerged (Szegedy et al., 2015; He et al., 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, 2018a).

We denote by $I_f$ the *domain of definition* of a function $f$ ("I" stands for "input") and by $O_f = f(I_f)$ its *image* ("O" stands for "output"), and we represent it as $I_f \xrightarrow{f} O_f$ or $f : I_f \to O_f$.

**Definition 16. Neural network (simply connected)**

Let $f$ be a function such that $I_f$ and $O_f$ are vector or tensor spaces.

$f$ is a *(simply connected) neural network function* if there are a series of affine functions $(g_k)_{k=1,2,..,L}$ and a series of non-linear derivable univariate functions $(h_k)_{k=1,2,..,L}$ such that:

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

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

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

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

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

where $b$ is a constant called *bias.*

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

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

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

**Examples**
Let $f : x \to y$ be a neural network. For example, if $f$ is used to classify its input $x$ in one of $c$ classes, then its output $y$ would be a vector of dimension $c$, and each dimension corresponds to a class. The prediction of $f$ for the class of $x$ is the dimension of $y$ where it has the bigger value. Typically, $f$ is terminated by a softmax activation (Wikipedia, 2018b), so that values of the

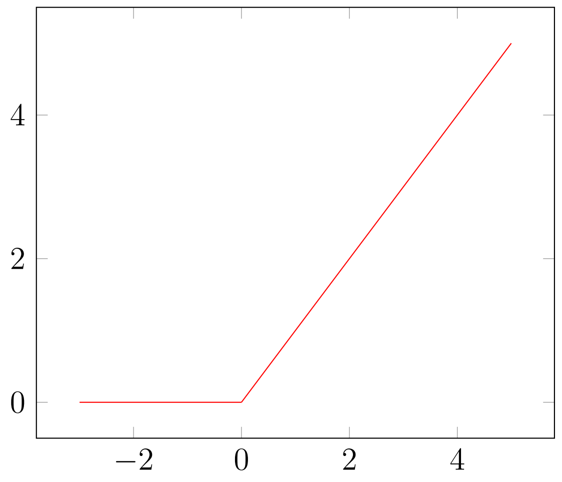

Figure 1: ReLU activation function

output $y$ fall in the range $[0, 1]$, and so that $y$ tends to have a dimension with a much bigger weight as to facilitates discrimination.

A neural network that comprises convolutional layers, *i.e.* layers *s.t.* $g$ is expressed with a convolution, is called a Convolutional Neural Network (CNN). A common example is the LeNet-5 architecture (LeCun et al., 1989) as depicted in Figure 2. It implements a function

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

where $g_1$ and $g_2$ are linear functions that applies 5x5 convolutions followed by subsampling, $h_1$, $h_2$ and $h_3$ are ReLU activations, and $h_4$ is a softmax activation. It was originally applied to the task of handwritten digit classifications (for example for automatically reading postal ZIP codes).

Another example is the VGG architecture, a very deep CNN, and was state-of-the-art in image classification in 2014 (Simonyan and Zisserman). It is depicted on Figure 3

In more recent years, state-of-the-art architecture can no longer be described with a simple formalization.

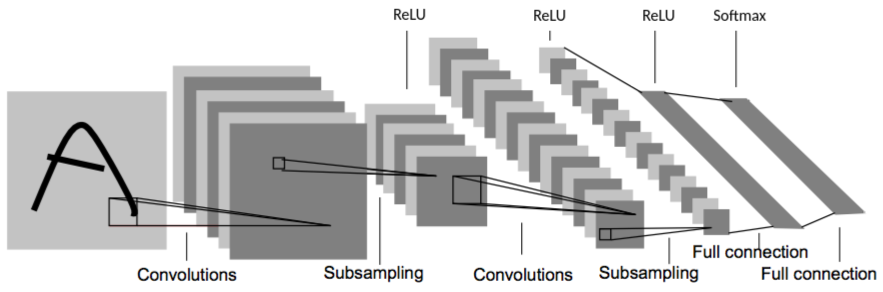

Figure 2: LeNet-5 (LeCun et al., 1989)

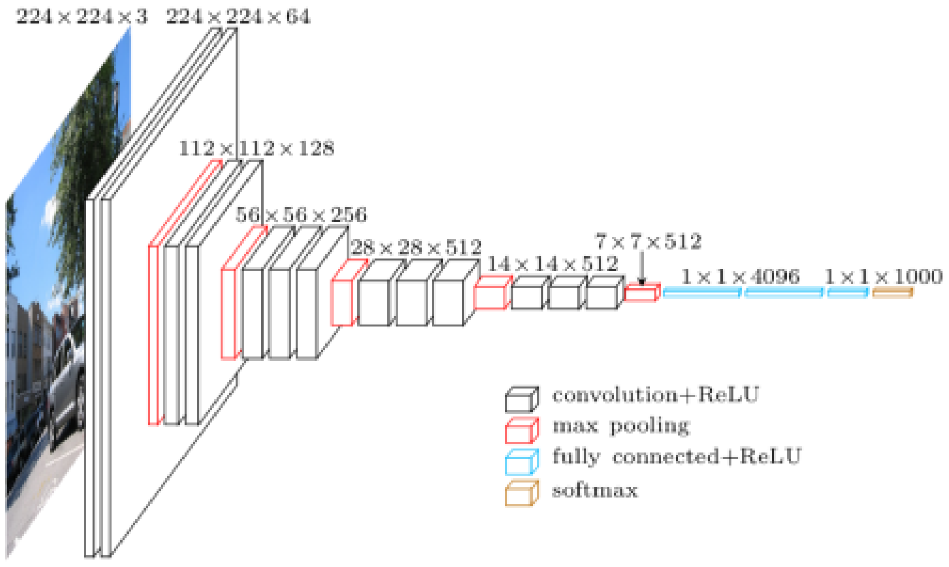

Figure 3: VGG-16 (Simonyan and Zisserman, 2014, figure from Cord, 2016)

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.

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

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

**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, as depicted on Figure 4. Densely connected networks (DenseNets, Huang et al., 2017) have their activations concatenated with all lower level activations. These neural networks had demonstrated state of the art performances on the imagenet classification challenge (Deng et al., 2009), outperforming simply connected neural networks. For example, DenseNet is depicted on Figure 5.

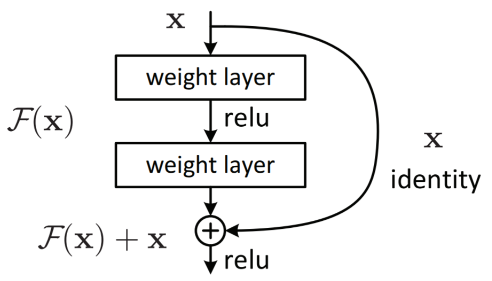
x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

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

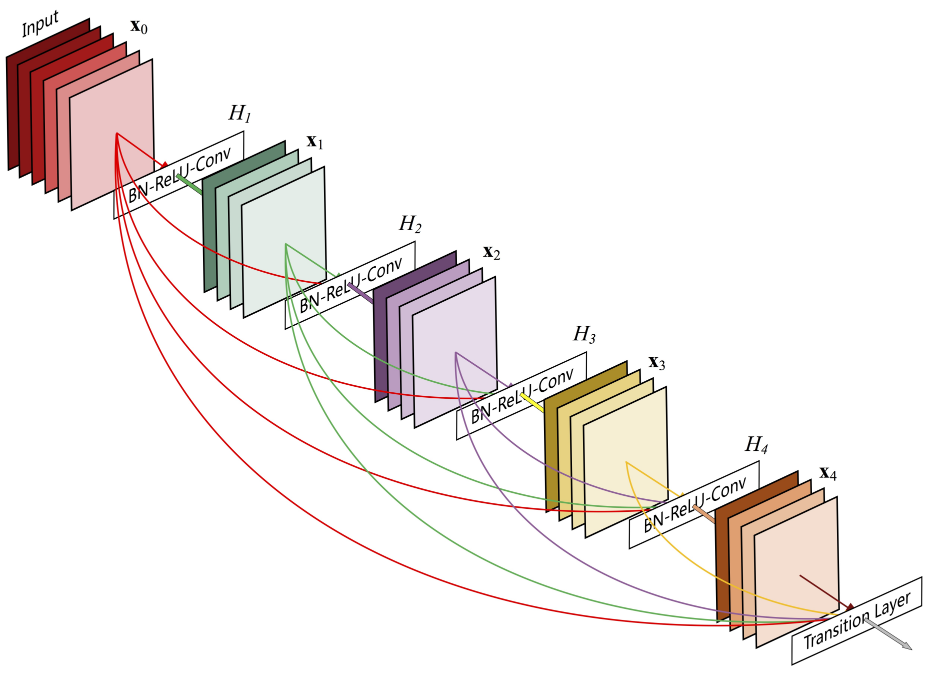

Figure 5: DenseNet (Huang et al., 2017)

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

### 1.2.2 Interpretation

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

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

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

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

#### Biological inspiration

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

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

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

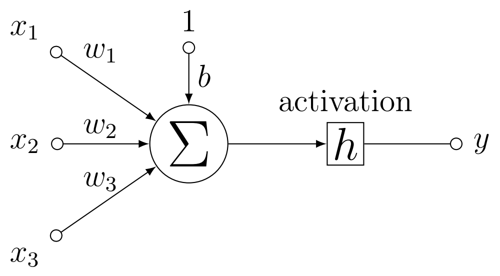

Figure 6: A neuron

### 1.2.3 Training

Given an objective function $F$, training is the process of incrementally modifying a neural network $f$ upon obtaining a better approximation of $F$. The most used training algorithms are based on gradient descent, as proposed in (Widrow and Hoff, 1960). These algorithms became popular since (Rumelhart et al., 1985). Informally, $f$ is parameterized with initial weights that characterize its linear parts. These weights are modified step by step. At each step, a batch of samples are fed to the network, and their approximation errors sum to a loss. The weights of the network are updated in the opposite direction to their gradient with respect to that loss. If the samples are shuffled and grouped in batches, this is called *Stochastic* gradient descent (SGD). Stochastic approximation (Robbins and Monro, 1985) tends to minimize effects of outliers on the training and is agnostic of the order in which the samples are fed.

**Definition 22. Weights**
Let consider the $k$-th layer of a neural network $f$. We define its weights as coordinates of a vector $\theta_k$, 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. $\theta_k$, denoted $\vec{\nabla}_{\theta_k}$, is used to update the weights via an optimization algorithm based on gradient descent and a learning rate $\alpha$, that is:

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

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

### Linear complexity

Without loss of generality, we assume that the neural network is simply connected. Thanks to the chain rule, $\vec{\nabla}_{\theta_k}$ can be computed using gradients that are w.r.t. $x_k$, denoted $\vec{\nabla}_{x_k}$, which in turn can be computed using gradients w.r.t. outputs of the next layer $k+1$, up to the gradients given on the output layer.

That is:

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

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

Obtaining,

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

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

### Backpropagation

We can remark that (5) rewrites as

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

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

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

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

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

### 1.2.4 Historical advances

#### Universal approximation

Early researches have shown that neural networks with one level of depth can approximate any real-valued function defined on a compact subset of $\mathbb{R}^n$. This result was first proved for sigmoidal activations (Cybenko, 1989), and then it was shown it did not depend on the sigmoidal activations (Hornik et al., 1989; Hornik, 1991).
For example, this result brings theoretical justification that objective functions exists (even though it 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 deep CNNs (Krizhevsky et al., 2012). The use of the fast ReLU activation function (Glorot et al., 2011) as well as leveraging graphical processing units with CUDA (Nickolls et al., 2008) were also key factors in overcoming this computational difficulty.

#### Adoption of ReLU activations

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

**Adoption of dropout**

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

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

**Rectifier neural netowrks**

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

**Benefits of depth**

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

**Benefits of branching operations**

Recent works have provided rationales supporting benefits of using branching operations, thus giving justifications for architectures obtained with the generic formalization. In particular, (Cohen et al., 2018) have analyzed the impact of residual connections used in Wavenet-like architectures (Van Den Oord et al., 2016) in terms of expressive efficiency, using tools from the field of tensor analysis ; (Orhan and Pitkow, 2018) have empirically demonstrated that skip connections can resolve some inefficiency problems inherent of fully-connected networks (dead activations, activations that are always equal, linearly dependent sets of activations).

### 1.2.5 Common layers

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

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

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

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

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

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

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

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

where $p$ and $q$ index slices along the last ranks.

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

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

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

**Benefits of convolutional layers**
Comparatively with dense layers, convolution layers enjoy a significant decrease in the number of weights. For example, an input $2 \times 2$ 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, Bengio, et al., 1995). A more thorough comparison and explanation of their assets will be discussed in Section **??**.

**Decrease of spatial dimensions**
Given a tensor input $x$, the $n$-dimensional convolutions between the inputs channels $x_p$ and slices of a weight tensor $w_{p,q}$ would result in outputs $y_q$ of shape $n_1^{(x)} - n_1^{(w)} + 1 \times \ldots \times n_n^{(x)} - n_n^{(w)} + 1$. So, in order to preserve shapes, a padding operation must pad $x$ with $n_1^{(w)} - 1 \times \ldots \times n_n^{(w)} - 1$ zeros beforehand. For example, the padding function of the library *tensorflow* (Abadi et al.,

2015) pads each rank with a balanced number of zeros on the left and right indices (except if $n_t^{(w)} - 1$ is odd then there is one more zero on the left).

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

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

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

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

Layers with stride or pooling downscale the signals that passes through the layer. These types of layers allows to compute features at a coarser level, giving the intuition that the deeper a layer is in the network, the more abstract is the information captured by the weights of the layer.

### A simple result

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

**Proposition 31. Connectivity matrix of a convolution with padding** A convolutional layer with padding $(g, h)$ is equivalently defined as its connectivity matrix $W_g$ being a $n_i \times n_o$ block matrix such that its blocks are Toeplitz matrices, and where each block corresponds to a couple $(p, q)$ of input channel $p$ and feature map $q$.

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

$$\begin{aligned} y &= (x *^n w)[j_1, \ldots, j_n] \\ &= \sum_{k_1=1}^{n_1^{(w)}} \cdots \sum_{k_n=1}^{n_n^{(w)}} x[j_1 + n_1^{(w)} - k_1, \ldots, j_n + n_n^{(w)} - k_n]\, w[k_1, \ldots, k_n] \\ &= \sum_{i_1=j_1}^{j_1+n_1^{(w)}-1} \cdots \sum_{i_n=j_n}^{j_n+n_n^{(w)}-1} x[i_1, \ldots, i_n]\, w[j_1 + n_1^{(w)} - i_1, \ldots, j_n + n_n^{(w)} - i_n] \\ &= \sum_{i_1=1}^{n_1^{(x)}} \cdots \sum_{i_n=1}^{n_n^{(x)}} x[i_1, \ldots, i_n]\, \widetilde{w}[i_1, j_1, \ldots, i_n, j_n] \\ &\text{where } \widetilde{w}[i_1, j_1, \ldots, i_n, j_n] = \\ &\qquad \begin{cases} w[j_1 + n_1^{(w)} - i_1, \ldots, j_n + n_n^{(w)} - i_n] & \text{if } \forall t, 0 \le i_t - j_t \le n_t^{(w)} - 1 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

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 $g_i(i_1, i_2, \dots, i_n) = g_j(i_1 - o_1, i_2 - o_2, \dots, i_n - o_n)$ defined if and only if $\forall t, 1 + o_t \leq i_t \leq n_t^{(y)}$, where the $\{o_t\}$ are the starting offsets of the non zero-padded values. $\widetilde{w} \mapsto W$ is reshaped by using $g_j$ for its covariant indices, and $g_i$ for its contravariant indices. The entries lost by using $g_i$ do not matter because they would have been nullified by the resulting matrix product. We remark that $W$ is exactly the block $(p, q)$ of $W_g$ (and not of $W_{g'}$). Now let's prove that it is a Toeplitz matrix.

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

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

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

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

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

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

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

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

## 1.3 Deep learning on graphs

### 1.3.1 Graph and signals

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

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

The terms *vertex* and *node* are used interchangeably. Additionaly, we consider that a graph is always *simple i.e.* no two edges share the same set of vertices. Unless stated otherwise, a graph is undirected, *i.e.* $(u, v)$ and $(v, u)$ refer to the same edge. When it's not the case, it is called a *digraph*. We define the relation $u \sim v \Leftrightarrow (u, v) \in E$. We precise the graph if needed over the symbol $\overset{G}{\sim}$. A *path* is a sequence $v_1 \sim \cdots \sim v_r$. A graph is said to be *connected* if there exists a path from any vertex to any other vertex. We define the *neighborhood* of a vertex as $\mathcal{N}_u = \{v \in V, u \sim v\}$. For digraphs, it is equal to the union of the *in*- and *out*-neighborhoods. We only consider graphs without isolated vertex (a vertex with an empty neighborhood). We also only consider *weighted* graphs. That is, a graph $G = \langle V, E\rangle$ is associated with a weight mapping $w : V^2 \to \mathbb{R}+$ *s.t.* $w(u, v) = 0 \Leftrightarrow u \nsim v$. If $G$ is finite, its *adjacency matrix* $A \in \mathbb{R}^{V\times V}$ is defined *w.r.t.* to a vertex ordering $V = \{v_1, \ldots, v_n\}$ as $A[i, j] = w(v_i, v_j)$. Figure 7 illustrates an example of a graph and its adjacency matrix.
The *order* of $G$ is equal to its number of vertices, possibly infinite. The *degree* of a vertex $v$ is equal to the number of edges it is attached to. For digraphs the degree is the sum of the *in*- and *out*-degrees. The *degree* of $G$ refers to its max degree. $G$ is said to be *degree-regular* if all its vertices have the same degree. If it is finite, its *degree matrix* $D$ (*w.r.t.* to a vertex ordering $V = \{v_1, \ldots, v_n\}$) is the diagonal matrix for which the diagonal entry corresponding to a vertex is the sum of the weights of the edges it is

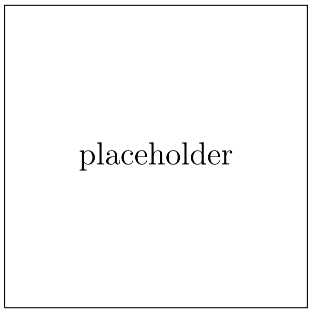

Figure 7: Example of a graph

part of. Its *laplacian matrix* $L$ is the substraction $L = D - A$, which can be *normalized* $L = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$, *left-normalized* $L = I - D^{-1}A$, or *right-normalized* $L = I - AD^{-1}$. A subgraph of $G$ induced by a subset $U \subset V$ is the graph with vertex and edge set restricted by $U$. The *complement* graph $G^C$ shares the same vertex set but $u \overset{G^C}{\sim} v \Leftrightarrow u \overset{G}{\nsim} v$. A *complete* graph is such that there exists an edge between any two vertices.

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

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

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

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

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

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

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

A *graph signal* on a graph $G = \langle V, E \rangle$ is a signal on its vertex set $V$. We denote by $\mathcal{S}(G)$ or $\mathcal{S}(V)$ the graph signal space. $G$ can be referred as the *underlying structure* of $\mathcal{S}(V)$. An *entry* of a signal $s$ is an image by $s$ of some $v \in V$ and we denote $s[v]$. If $v$ is represented by an $n$-tuple, we can also write $s[v_1, v_2, \dots, v_n]$. The *support* of a signal $s \in \mathcal{S}(V)$ is the subset $\operatorname{supp}(s) \subset V$ on which $s \neq 0$. For spaces of signals that aren't real-valued, their codomain $\mathbb{E}$ is precised in the subscript $\mathcal{S}_{\mathbb{E}}(V)$.

### 1.3.2 Learning tasks

There are many tasks related to deep learning on graphs.

**Supervised classification of graph signals**
This is the classical application of deep learning transposed to graph signals, rather than image or audio signals. It is the principled targeted task we will have in mind in the course of the remainder of this manuscript. Given a graph $G = \langle V, E \rangle$ and an input signal $x \in \mathcal{S}(G)$ the goal is to classify $x$. If there are $c$ possible classes, a neural network $f$ outputs a vector $y = f(x)$ of dimension $c$, and its dimension with the biggest weight determines the predicted class. Indeed, a standard MLP can be trained on a dataset of graph signals. However, an MLP wouldn't take the graph structure $G$ into consideration. By similarity with CNNs that leverage the grid structure of images to achieve better performances than MLPs, a challenge is to define a neural network on graph signals that can leverage $G$. We review some models from the litterature in Section 1.3.3 and in Section 1.3.4. We develop

an algebraic understanding in Chapter **??** of why and how they should work, and also propose our own models and point of view in Chapter **??**.

### Semi-supervised classification of nodes

This task is in some way obtained from a transposed perspective of the previous one. Given a dataset of graph signals, represented as a matrix $X \in \mathbb{R}^{n \times N}$, where the rows represent the nodes, and the columns represent the signals, the goal is to classify the nodes. This amounts to classify the rows, whereas the previous task amounts to classify the columns. As opposed to the previous one, this task is *transductive i.e.* node data from the test set are available during training (but their labels are not), and it is *semi-supervised i.e.* some node labels of the train set are unknown. This allows to learn on much more data than if we were restricted to labeled data. In this task, the edges connect learning samples, however in the previous one, the edges were connecting features of learning samples. This is this edge relationship between learning samples that renders the semi-supervised approach possible. This task have received much more attention than the previous one in the recent litterature. We explain why in Section 1.3.3.

### Other learning tasks

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

### 1.3.3 Spectral methods

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

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

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

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

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

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

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

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

And so,

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

That is, $s^T Ls$ is some sort of measure of *smoothness* of the signal $s$, penalized by the weights $w$. The bigger is $w(u,v)$, the closest $s(u)$ and $s(v)$ must be to lower the smoothness (17). Since $L$ is symmetric, its eigenvalues are non-negative real numbers, and $U$ diagonalizes $L$ as $\Lambda = ULU^*$. Denote $(\lambda_i)_i$ the eigenvalues, the smoothness measure rewrites:

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

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

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

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

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

$$\begin{aligned} g_\theta(L)s &= g_\theta(U^* \Lambda U)s = U^* g_\theta(\Lambda) U s \\ &= \widetilde{g_\theta(\Lambda)}\mathbf{1} * s \end{aligned} \tag{20}$$

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

However, it is hard to evaluate if a model performs well on the task of supervised classification of graph signals, because there are not much known

datasets in the litterature for which the given graph domain holds enough information.
For example, Defferrard et al. built a graph signal dataset from a text categorization dataset called 20NEWS (Joachims, 1996). Each text is represented as a word2vec vector, and features are linked by edges with their nearest neighbors. However, their model (ChebNet32) fails to surpass Multinomial Naive Bayes (MNB). Moreover, even though they report that their model beat MLPs, our experiments show the contrary. In results we report in Table 1, we see that a lighter MLP, composed of a single Fully-Connected (FC) layer with ReLU and 20% dropout outperforms ChebNet32. We replicated their preprocessing phase from the code on their github repository and averaged our results on 10 runs of 20 epochs.

| MNB | FC2500 | FC2500-FC500 | ChebNet32 | FC500 |
|---|---|---|---|---|
| 68.51%[a] | 64.64%[a] | 65.76%[a] | 68.26%[a] | **71.46**±0.08%[b] |

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

Table 1: Accuracies on 20NEWS

Despite the significant theoretical contribution, this negative result stresses out the importance of the practical graph used to support the convolution, a point that they also discussed. Henaff et al., 2015 proposed supervised graph estimation techniques, but a better graph signal dataset would be one that come with an already suitable graph, that of current literature is still lacking.
On the other hand, attention in the domain has shifted toward the task of semi-supervised classification of nodes, where good datasets are not lacking. For example, Levie et al., 2017 mainly demonstrate the usefulness of their model on these type of tasks. They define polynomial filters, for which

Chebychev filters are a special case, that are capable to specialize in narrow bands of frequency in the spectral domain.

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

### 1.3.4 Vertex-domain methods

As their name suggests, vertex-domain methods operates directly on the vertices of the graph. These works were originally motivated by chemistry datasets (Duvenaud et al., 2015; Kearnes et al., 2016). Convolution is defined as a function $f$ of the kernel weights $\theta$ and neighboring vertices (contained in the receptive field $\mathcal{R}(v)$), usually based on dot products. That is

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

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

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

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

where $X \in \mathbb{R}^{n \times N}$, $\Theta \in \mathbb{R}^{N \times M}$, $n$ is the number of nodes, $N$ is the number of input channels (features per node), and $M$ is the number of output feature maps. On the left, powers of $\widetilde{L}$ diffuse the graph signal $X$ to share node information. On the right, $\Theta$ maps the diffused signals to another representation. So in essence, this formulation is more a vertex-domain method. They found that the best performing filters were expressed in a simplified form

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

where $\widetilde{A}$ is the normalized adjacency matrix of the graph to which self-loops are added. They called the architecture composed with these simple filters a Graph Convolution Network (GCN). Similiarly, $AX$ shares node information via the edges of the graph and $\Theta$ makes the model learns. This fomulation attracted a lot of research attention and was, in particular, extended with attention mechanism (no pun intended), inspired from the field of neural machine translation (Bahdanau et al., 2014). Attention can be learned toward which input feature map is most useful (Velickovic et al., 2017), or which neighboring vertex is (Lee et al., 2018). Works extending GCN are numerous in recent days (*e.g.* Niepert and Garcia-Duran, 2018), and covering them all would be frivolous, especially considering that their novelty is limited and often specialized to use cases of particular datasets.

# 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 p. 27).

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. 25).

Atwood, James and Don Towsley (2016). "Diffusion-convolutional neural networks". In: *Advances in Neural Information Processing Systems*, pp. 1993–2001 (cit. on p. 40).

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio (2014). "Neural machine translation by jointly learning to align and translate". In: *arXiv preprint arXiv:1409.0473* (cit. on p. 41).

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

Bengio, Yoshua (2009). “Learning deep architectures for AI”. In: *Foundations and trends® in Machine Learning* 2.1, pp. 1–127 (cit. on p. 23).

Bianchini, Monica and Franco Scarselli (2014). “On the complexity of neural network classifiers: A comparison between shallow and deep architectures”. In: *IEEE transactions on neural networks and learning systems* 25.8, pp. 1553–1565 (cit. on p. 25).

Bruna, Joan and Stéphane Mallat (2013). “Invariant scattering convolution networks”. In: *IEEE transactions on pattern analysis and machine intelligence* 35.8, pp. 1872–1886 (cit. on p. 40).

Bruna, Joan, Wojciech Zaremba, Arthur Szlam, and Yann LeCun (2013). “Spectral networks and locally connected networks on graphs”. In: *arXiv preprint arXiv:1312.6203* (cit. on p. 36).

Chung, Fan R. K. (1996). *Spectral Graph Theory (CBMS Regional Conference Series in Mathematics, No. 92)*. American Mathematical Society. ISBN: 0821803158 (cit. on p. 36).

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. 24).

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. 25).

Cohen, Nadav, Ronen Tamari, and Amnon Shashua (2018). “Boosting Dilated Convolutional Networks with Mixed Tensor Decompositions”. In: *International Conference on Learning Representations*. URL: `https://openreview.net/forum?id=S1JHhv6TW` (cit. on pp. 24, 25).

Cord, Matthieu (2016). *Deep learning an weak supervision for image classification*. [Online; accessed April-2018]. URL: `http://webia.lip6.fr/~cord/pdfs/news/TalkDeepCordI3S.pdf` (cit. on p. 16).

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. 23).

Defferrard, Michaël, Xavier Bresson, and Pierre Vandergheynst (2016). "Convolutional neural networks on graphs with fast localized spectral filtering". In: *Advances in Neural Information Processing Systems*, pp. 3837–3845 (cit. on pp. 38, 39).

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. 24, 25).

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. 17, 23).

Du, Jian, Shanghang Zhang, Guanhang Wu, José MF Moura, and Soummya Kar (2017). "Topology adaptive graph convolutional networks". In: *arXiv preprint arXiv:1710.10370* (cit. on p. 40).

Duvenaud, David K, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P Adams (2015). "Convolutional Networks on Graphs for Learning Molecular Fingerprints". In: *Advances in Neural Information Processing Systems*, pp. 2215–2223 (cit. on p. 40).

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. 25).

Gama, Fernando, Alejandro Ribeiro, and Joan Bruna (2018). "Diffusion Scattering Transforms on Graphs". In: *arXiv preprint arXiv:1806.08829* (cit. on p. 40).

Gilmer, Justin, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl (2017). “Neural message passing for quantum chemistry”. In: *arXiv preprint arXiv:1704.01212* (cit. on p. 40).

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 p. 23).

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. 14, 23).

Grover, Aditya and Jure Leskovec (2016). “node2vec: Scalable feature learning for networks”. In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, pp. 855–864 (cit. on p. 35).

Hackbusch, Wolfgang (2012). *Tensor spaces and numerical tensor calculus.* Vol. 42. Springer Science & Business Media (cit. on pp. 5, 6).

Hamilton, Will, Zhitao Ying, and Jure Leskovec (2017a). “Inductive representation learning on large graphs”. In: *Advances in Neural Information Processing Systems*, pp. 1024–1034 (cit. on p. 35).

Hamilton, William L, Rex Ying, and Jure Leskovec (2017b). “Representation learning on graphs: Methods and applications”. In: *arXiv preprint arXiv:1709.05584* (cit. on p. 35).

Hammond, David K, Pierre Vandergheynst, and Rémi Gribonval (2011). “Wavelets on graphs via spectral graph theory”. In: *Applied and Computational Harmonic Analysis* 30.2, pp. 129–150 (cit. on pp. 36, 38, 40).

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. 24).

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. 24).

— (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. 13, 17, 18).

Hechtlinger, Yotam, Purvasha Chakravarti, and Jining Qin (2017). "A generalization of convolutional neural networks to graph-structured data". In: *arXiv preprint arXiv:1704.08165* (cit. on p. 40).

Henaff, Mikael, Joan Bruna, and Yann LeCun (2015). "Deep convolutional networks on graph-structured data". In: *arXiv preprint arXiv:1506.05163* (cit. on p. 39).

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. 13).

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. 23).

Hornik, Kurt (1991). "Approximation capabilities of multilayer feedforward networks". In: *Neural networks* 4.2, pp. 251–257 (cit. on p. 23).

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. 23, 26).

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. 13, 17, 18).

Jaderberg, Max, Karen Simonyan, Andrew Zisserman, et al. (2015). "Spatial transformer networks". In: *Advances in neural information processing systems*, pp. 2017–2025 (cit. on p. 38).

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. 23).

Joachims, Thorsten (1996). *A Probabilistic Analysis of the Rocchio Algorithm with TFIDF for Text Categorization.* Tech. rep. Carnegie-mellon univ pittsburgh pa dept of computer science (cit. on p. 39).

Kearnes, Steven, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley (2016). "Molecular graph convolutions: moving beyond fingerprints". In: *Journal of computer-aided molecular design* 30.8, pp. 595–608 (cit. on p. 40).

Kipf, Thomas N and Max Welling (2016). "Semi-supervised classification with graph convolutional networks". In: *arXiv preprint arXiv:1609.02907* (cit. on p. 40).

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. 24).

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. 13, 23, 27).

LeCun, Y. (1987). "Modeles connexionnistes de l'apprentissage (connectionist learning models)". PhD thesis. Université P. et M. Curie (Paris 6) (cit. on p. 19).

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. 13, 27).

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. 13, 15, 16, 23).

Lee, John Boaz, Ryan A Rossi, Sungchul Kim, Nesreen K Ahmed, and Eunyee Koh (2018). "Attention Models in Graphs: A Survey". In: *arXiv preprint arXiv:1807.07984* (cit. on pp. 40, 41).

Levie, Ron, Federico Monti, Xavier Bresson, and Michael M Bronstein (2017). "CayleyNets: Graph Convolutional Neural Networks with Complex Rational Spectral Filters". In: *arXiv preprint arXiv:1705.07664* (cit. on p. 39).

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. 25).

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. 24).

Marcus, Marvin (1975). "Finite dimensional multilinear algebra". In: (cit. on p. 5).

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133 (cit. on p. 19).

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. 25).

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean (2013a). "Distributed representations of words and phrases and their compositionality". In: *Advances in neural information processing systems*, pp. 3111–3119 (cit. on p. 35).

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean (2013b). "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (cit. on p. 35).

Monti, Federico, Davide Boscaini, Jonathan Masci, Emanuele Rodolà, Jan Svoboda, and Michael M Bronstein (2016). "Geometric deep learning on graphs and manifolds using mixture model CNNs". In: *arXiv preprint arXiv:1611.08402* (cit. on p. 40).

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. 24).

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. 23).

Niepert, Mathias and Alberto Garcia-Duran (2018). "Towards a Spectrum of Graph Convolutional Networks". In: *arXiv preprint arXiv:1805.01837* (cit. on p. 41).

Niepert, Mathias, Mohamed Ahmed, and Konstantin Kutzkov (2016). "Learning Convolutional Neural Networks for Graphs". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, pp. 2014–2023 (cit. on pp. 35, 40).

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. 25).

Pan, Xingyuan and Vivek Srikumar (2016). "Expressiveness of rectifier networks". In: *International Conference on Machine Learning*, pp. 2427–2435 (cit. on p. 25).

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 p. 24).

Pasdeloup, Bastien, Vincent Gripon, Jean-Charles Vialatte, and Dominique Pastor (2017). "Convolutional neural networks on irregular domains through approximate translations on inferred graphs". In: *arXiv preprint arXiv:1710.10035* (cit. on p. 40).

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. 25).

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. 25).

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. 25).

Robbins, Herbert and Sutton Monro (1985). "A stochastic approximation method". In: *Herbert Robbins Selected Papers.* Springer, pp. 102–109 (cit. on p. 20).

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. 13, 20).

Sankar, Aravind, Xinyang Zhang, and Kevin Chen-Chuan Chang (2017). "Motif-based Convolutional Neural Network on Graphs". In: *arXiv preprint arXiv:1711.05697* (cit. on p. 40).

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. 36).

Simonovsky, Martin and Nikos Komodakis (2017). "Dynamic Edge-Conditioned Filters in Convolutional Neural Networks on Graphs". In: *arXiv preprint arXiv:1704.02901* (cit. on p. 40).

Simonyan, Karen and Andrew Zisserman (2014). "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (cit. on pp. 13, 15, 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 p. 24).

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. 13, 17).

Tixier, Antoine Jean-Pierre, Giannis Nikolentzos, Polykarpos Meladianos, and Michalis Vazirgiannis (2017). "Classifying Graphs as Images with Convolutional Neural Networks". In: *arXiv preprint arXiv:1708.02218* (cit. on p. 35).

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. 25).

Velickovic, Petar, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio (2017). "Graph Attention Networks". In: *stat* 1050, p. 20 (cit. on pp. 40, 41).

Vialatte, Jean-Charles, Vincent Gripon, and Grégoire Mercier (2016). "Generalizing the convolution operator to extend cnns to irregular domains". In: *arXiv preprint arXiv:1606.01166* (cit. on p. 40).

Vialatte, Jean-Charles, Vincent Gripon, and Gilles Coppin (2017). "Learning Local Receptive Fields and their Weight Sharing Scheme on Graphs". In: *arXiv preprint arXiv:1706.02684* (cit. on p. 40).

Widrow, Bernard and Marcian E Hoff (1960). *Adaptive switching circuits.* Tech. rep. STANFORD UNIV CA STANFORD ELECTRONICS LABS (cit. on p. 20).

Wikipedia, contributors (2018a). *Feedforward neural network — Wikipedia, The Free Encyclopedia.* [Online; accessed April-2018]. URL: `https://en.wikipedia.org/wiki/Feedforward_neural_network` (cit. on p. 13).

— (2018b). *Softmax function — Wikipedia, The Free Encyclopedia.* [Online; accessed April-2018]. URL: `https://en.wikipedia.org/wiki/Softmax_function` (cit. on p. 14).

Williamson, S Gill (2015). "Tensor spaces-the basics". In: *arXiv preprint arXiv:1510.02428* (cit. on p. 5).

Yi, Li, Hao Su, Xingwen Guo, and Leonidas Guibas (2016). "Syncspeccnn: Synchronized spectral CNN for 3d shape segmentation". In: *arXiv preprint arXiv:1612.00606* (cit. on p. 38).

Zell, Andreas (1994). *Simulation neuronaler netze.* Vol. 1. Addison-Wesley Bonn (cit. on p. 13).

Zoph, Barret and Quoc V Le (2016). "Neural architecture search with reinforcement learning". In: *arXiv preprint arXiv:1611.01578* (cit. on p. 13).

Zou, D. and G. Lerman (2018). "Graph Convolutional Neural Networks via Scattering". In: *ArXiv e-prints.* arXiv: `1804.00099 [cs.IT]` (cit. on p. 40).