

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	Neural Networks	13
1.2.1	Simple formalization	13
1.2.2	Generic formalization	16
1.2.3	Interpretation	18
1.2.4	Parameterization and training	19
1.2.5	Examples of layer	23
1.2.6	Examples of regularization	28
1.2.7	Examples of architecture	28
1.3	Graphs and signals	29
1.3.1	Basic definitions	29
1.3.1.1	Graphs	29
1.3.1.2	(Real-valued) Signals	32
1.3.2	Graphs in deep learning	33
1.3.2.1	Connectivity graph	34
1.3.2.2	Computation graph	35
1.3.2.3	Underlying graph structure and signals	35
1.3.2.4	Graph-structured dataset	35

3	Application to neural networks on graph domains	37
3.1	Layer representations	39
3.1.1	Neural interpretation of tensor spaces	39
3.1.2	Graph structure of the propagation	40
3.1.3	Underlying graph structure	41
	Bibliography	43

Chapter 1

Presentation of the field

Contents

1.1	Tensors	5
1.1.1	Definition	5
1.1.2	Manipulation	7
1.1.3	Binary operations	9
1.2	Neural Networks	13
1.2.1	Simple formalization	13
1.2.2	Generic formalization	16
1.2.3	Interpretation	18
1.2.4	Parameterization and training	19
1.2.5	Examples of layer	23
1.2.6	Examples of regularization	28
1.2.7	Examples of architecture	28
1.3	Graphs and signals	29
1.3.1	Basic definitions	29
1.3.2	Graphs in deep learning	33

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 $\mathbb{V}_1, \mathbb{V}_2, \dots, \mathbb{V}_r$, 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 $n_1^{(t)} \times n_2^{(t)} \times \dots \times n_r^{(t)}$.

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 $\mathbb{I} = \prod_{k=1}^r \llbracket 1, n_k \rrbracket$ 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 $i_k \in \llbracket 1, n_k^{(t)} \rrbracket$ 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 $s : \mathbb{T} = \bigotimes_{k=1}^r \mathbb{V}_k \rightarrow \bigotimes_{k=1}^{r-s} \mathbb{V}_k$, 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 $f : \mathbb{T} \rightarrow \mathbb{V}$, 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 $g : \prod_{k=1}^r \llbracket 1, n_k \rrbracket \rightarrow \llbracket 1, n \rrbracket$ 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

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

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 $n_1^{(1)} \times n_2^{(1)} \times \dots \times n_{r_1}^{(1)}$, and \mathbb{T}_2 a tensor space of shape $n_1^{(2)} \times n_2^{(2)} \times \dots \times n_{r_2}^{(2)}$, such that $\forall k \in \llbracket 1, s \rrbracket, n_{r_1-(s-k)}^{(1)} = n_k^{(2)}$, then the tensor contraction between $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$ is defined as:

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

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

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:

$$t_{1,i_1^{(1)} \dots i_{r_1-s}^{(1)}}{}^{k_1 \dots k_s} t_{2,k_1 \dots k_s}{}^{i_{s+1}^{(2)} \dots i_{r_2}^{(2)}} = t_{3,i_1^{(1)} \dots i_{r_1-s}^{(1)}}{}^{i_{s+1}^{(2)} \dots i_{r_2}^{(2)}} \quad (2)$$

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

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

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

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 $\forall p \in \llbracket 1, n \rrbracket, n_p^{(1)} \geq n_p^{(2)}$, is defined as:

$$\left\{ \begin{array}{l} t_1 *^n t_2 = t_3 \in \mathbb{T}_3 \text{ of shape } n_1^{(3)} \times \cdots \times n_n^{(3)} \text{ where} \\ \forall p \in \llbracket 1, n \rrbracket, n_p^{(3)} = n_p^{(1)} - n_p^{(2)} + 1 \\ t_3[i_1, \dots, i_n] = \sum_{k_1=1}^{n_1^{(2)}} \cdots \sum_{k_n=1}^{n_n^{(2)}} t_1[i_1 + n_1^{(2)} - k_1, \dots, i_n + n_n^{(2)} - k_n] t_2[k_1, \dots, k_n] \end{array} \right.$$

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. \square

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 $\forall p \in \llbracket 1, n \rrbracket, n_p^{(1)} \geq n_p^{(2)}$, is defined as:

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

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, Bengio, et al., 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, 2018).

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 $(g_k)_{k=1,2,\dots,L}$ and a series of non-linear derivable univariate functions $(h_k)_{k=1,2,\dots,L}$ such that:

$$\left\{ \begin{array}{l} \forall k \in \llbracket 1, L \rrbracket, f_k = h_k \circ g_k, \\ I_f = I_{f_1} \xrightarrow{f_1} O_{f_1} \cong I_{f_2} \xrightarrow{f_2} \dots \xrightarrow{f_L} O_{f_L} = O_f, \\ f = f_L \circ \dots \circ f_2 \circ f_1 \end{array} \right.$$

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 $x_k = f_k \circ \dots \circ f_2 \circ f_1(x)$ 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., 2018).

Rectifier neural networks

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 \tilde{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:
 $f_1, f_2, \dots, f_n \in \mathcal{N} \Rightarrow f_1 \bowtie f_2 \bowtie \cdots \bowtie f_n \in \mathcal{N}$

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. 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., 2018) 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.

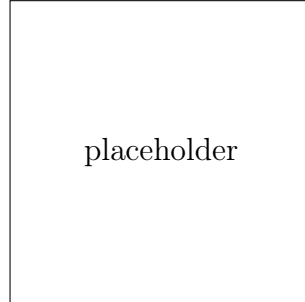


Figure 1.1: A neuron

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:

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

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:

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

$$\begin{aligned} \vec{\nabla}_{x_k} &= J_{x_k}(x_{k+1}) \vec{\nabla}_{x_{k+1}} \\ \vec{\nabla}_{x_{k+1}} &= J_{x_{k+1}}(x_{k+2}) \vec{\nabla}_{x_{k+2}} \end{aligned} \quad (5)$$

...

$$\vec{\nabla}_{x_{L-1}} = J_{x_{L-1}}(x_L) \vec{\nabla}_{x_L}$$

Obtaining,

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

where $J_{\text{wrt}}(\cdot)$ 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 $\vec{\nabla}_{x_k} = (\tilde{h}_k \circ \tilde{g}_k)(\vec{\nabla}_{x_{k+1}})$ 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 *sparingly 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 of $n_i \times n_o$ 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 $n_1^{(x)} - n_1^{(w)} + 1 \times \dots \times n_n^{(x)} - n_n^{(w)} + 1$. So, in order to preserve shapes, a padding operation must pad x with $n_1^{(w)} - 1 \times \dots \times n_n^{(w)} - 1$ 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

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

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

$$y_{j_1 \dots j_n} = x_{i_1 \dots i_n} \tilde{w}^{i_1 \dots i_n}{}_{j_1 \dots j_n} \quad (10)$$

Following Proposition 11, we reshape (10) as a matrix product. To reshape $y \mapsto Y$, we use the row major order bijections g_j as in (1) defined onto $\{(j_1, \dots, j_n), \forall t, 1 \leq j_t \leq n_t^{(y)}\}$. To reshape $x \mapsto X$, we use the same row major order bijection g_j , however defined on the indices that support non zero-padded values, so that zero-padded values are lost after reshaping. That is, we use a bijection g_i such that $g_i(i_1, i_2, \dots, i_n) = g_j(i_1 - o_1, i_2 - o_2, \dots, i_n - o_n)$ defined if and only if $\forall t, 1 + o_t \leq i_t \leq n_t^{(y)}$, where the $\{o_t\}$ are the starting offsets of the non zero-padded values. $\tilde{w} \mapsto W$ 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

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

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

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

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. \square

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, Bengio, et al., 1995). A more thorough comparison and explanation of their assets will be discussed in Section ??.

Definition 31. Stride

A convolutional layer with *stride* is a convolutional layer that computes strided convolutions (with $\text{stride} > 1$) 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., 1998) 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 $w : E \rightarrow \mathbb{R}^*$.

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

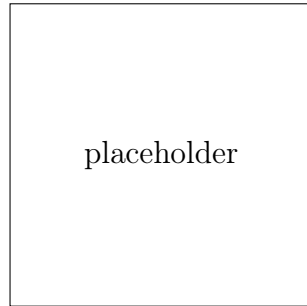


Figure 1.2: Example of a graph

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

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

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 $w : E \rightarrow \mathbb{R}^*$. 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* $\tilde{G} = \langle \tilde{V}, \tilde{E} \rangle$ of a graph $G = \langle V, E \rangle$, *induced* by $\tilde{V} \subseteq V$, is such that $\forall (u, v) \in \tilde{V}^2, u \tilde{\sim} v \Leftrightarrow u \stackrel{G}{\sim} v$.

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:

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

A (*rectangular*) *grid graph* of size $n \times m$ is the subgraph of a grid graph induced by $\llbracket 1, n \rrbracket \times \llbracket 1, m \rrbracket$.

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 $s : V \rightarrow \mathbb{R}$.

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 $s[v_1, v_2, \dots, v_n]$.

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.

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 Θ_{ij} are taking their values only into the finite set $K = \{w_1, w_2, \dots, w_\kappa\}$ of size κ , which we will refer to as the *kernel*

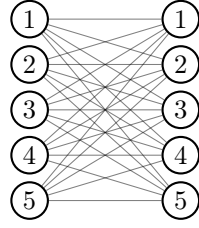


Figure 1.3: Examples

of *weights*. Then we can define a labelling of the edges $s : E \rightarrow K$. 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 1.4 depicts the connectivity graph of a 1-d convolution layer and its weight sharing scheme.

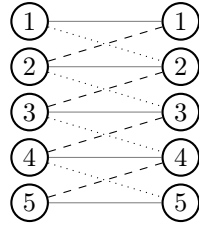


Figure 1.4: Depiction of a 1D-convolutional layer and its weight sharing scheme.

TODO: Add weight sharing scheme in Figure 1.4

1.3.2.2 Computation graph

1.3.2.3 Underlying graph structure and signals

1.3.2.4 Graph-structured dataset

Chapter 3

Application to neural networks on graph domains

Contents

3.1	Layer representations	39
3.1.1	Neural interpretation of tensor spaces	39
3.1.2	Graph structure of the propagation	40
3.1.3	Underlying graph structure	41

Introduction

TODO:

3.1 Layer representations

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

3.1.1 Neural interpretation of tensor spaces

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

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

More precisely, we have the following relation.

Lemma 49. Relation between tensor and signal spaces

Let \mathbb{V}, \mathbb{U} vector spaces, and \mathbf{v}, \mathbf{u} their canonical bases. Let \mathbb{T} a tensor space. \otimes and \times denote tensor and cartesian products. Then,

- (i) $\mathbb{V} \cong \mathcal{S}(\mathbf{v})$
- (ii) $\mathbb{V} \otimes \mathbb{U} \cong \mathcal{S}(\mathbf{v} \times \mathbf{u})$
- (iii) $\mathbb{V} \otimes \mathbb{T} \cong \mathcal{S}_{\mathbb{T}}(\mathbf{v})$

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

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

- (ii) $\tilde{x}[\mathbf{v}^i, \mathbf{u}^j] = x[i, j]$
- (iii) $\tilde{x}[\mathbf{v}^i] = x[i, :, \dots, :]$

□

Denote $V = \times_{k=1}^d \mathfrak{v}$ and $U = \times_{k=1}^e \mathfrak{u}$. Therefore, we can thus identify the input and output spaces as $I = \mathcal{S}(V) \otimes \bigotimes_{k=d+1}^p \mathbb{V}_k$ and $O = \mathcal{S}(U) \otimes \bigotimes_{l=e+1}^q \mathbb{U}_l$. As $\mathcal{S}(\mathfrak{v}) \otimes \mathbb{T} = \mathcal{S}_{\mathbb{T}}(\mathfrak{v})$, an object of V or U can be interpreted as the representation of a *neuron* which can take multiple values.

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

3.1.2 Graph structure of the propagation

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

Remark. Using the mapping defined in the proof of Lemma 49, for notational convenience, we'll abusively consider x as a vector (eventually reshaped from a tensor), and W as a binary tensor product for its indexing (*i.e.* $W[u, v] := W[i, j]$ where $u = \mathfrak{u}^i$ and $v = \mathfrak{v}^j$).

Definition 50. Propagation graph

The *propagation graph* P of a layer $\mathcal{L} = (W, h)$ is the bipartite graph that has the connectivity matrix W for bipartite adjacency matrix.

An example is depicted on Figure 3.1.

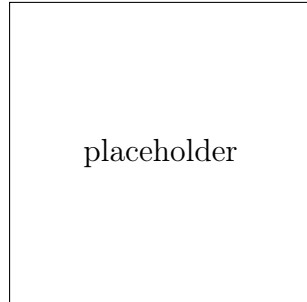


Figure 3.1: A propagation graph

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

Definition 51. Topological space

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

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

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

3.1.3 Underlying graph structure

If the input and output spaces of a layer are topologically isomorph (in the sense that the open sets and the neighborhoods are preserved by a bijective mapping *i.e.* $u \xleftarrow{P} v \Leftrightarrow \tilde{u} \xrightarrow{P} \tilde{v}$), then we can identify output neurons with input neurons $U := V$. In that case, the propagation is supported *on* an underlying graph structure, that is not a bipartite graph (contrary to the propagation graph).

Definition 52. Underlying graph structure

Let a layer $\mathcal{L} : \mathcal{S}(V) \rightarrow \mathcal{S}(U)$ such that $\mathcal{T}_V \cong \mathcal{T}_U$. Its *underlying graph structure* is the graph $G = \langle V, E \rangle$ such that $E = \{\{u, v\} \in V^2, \tilde{u} \in \mathcal{N}_v\}$.

TODO:

Figure ?? depicts a underlying graph structure basing a convolution, and the corresponding propagation graph.

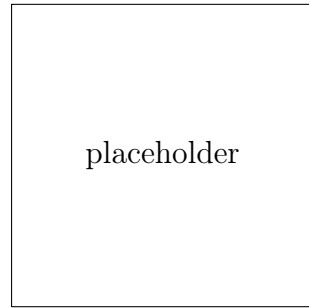


Figure 3.2: Underlying graph Vs Prop graph

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. 24).
- 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. 16).
- 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. 15).
- 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. 16).

- Bottou, Léon (2010). “Large-scale machine learning with stochastic gradient descent”. In: *Proceedings of COMPSTAT’2010*. Springer, pp. 177–186 (cit. on p. 28).
- 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. 14).
- 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. 16).
- 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. 15, 17).
- Cybenko, George (1989). “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of control, signals and systems* 2.4, pp. 303–314 (cit. on p. 14).
- 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. 15, 16).
- 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. 15, 17).
- 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. 16).
- 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. 15).

- 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, 15).
- Hackbusch, Wolfgang (2012). *Tensor spaces and numerical tensor calculus*. Vol. 42. Springer Science & Business Media (cit. on pp. 5, 6).
- 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. 15).
- 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. 14).
- (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).
- 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. 15).
- Hornik, Kurt (1991). “Approximation capabilities of multilayer feedforward networks”. In: *Neural networks* 4.2, pp. 251–257 (cit. on p. 14).
- 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. 14, 28).
- 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).
- 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. 14).
- 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. 14).
- 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, 15, 27).
- LeCun, Y. (1987). “Modeles connexionnistes de l’apprentissage (connectionist learning models)”. PhD thesis. Université P. et M. Curie (Paris 6) (cit. on p. 18).
- 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, 14).
- LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner (1998). “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11, pp. 2278–2324 (cit. on p. 28).

- 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. 16).
- 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. 14).
- 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. 18).
- 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. 16).
- 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. 16).
- 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. 15).
- 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. 18).
- Pan, Xingyuan and Vivek Srikumar (2016). “Expressiveness of rectifier networks”. In: *International Conference on Machine Learning*, pp. 2427–2435 (cit. on p. 16).
- 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. 15, 16).
- 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. 16).
- 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. 16).
- 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. 16).
- 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, 19, 20).
- Simonyan, Karen and Andrew Zisserman (2014). “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (cit. on p. 13).
- 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. 28).
- 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).
- 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. 17).
- Widrow, Bernard and Marcian E Hoff (1960). *Adaptive switching circuits*. Tech. rep. STANFORD UNIV CA STANFORD ELECTRONICS LABS (cit. on pp. 19, 20).
- Wikipedia, contributors (2018). *Feedforward neural network* — *Wikipedia, The Free Encyclopedia*. [Online; accessed April-2018]. URL: https://en.wikipedia.org/wiki/Feedforward_neural_network (cit. on p. 13).
- Williamson, S Gill (2015). “Tensor spaces-the basics”. In: *arXiv preprint arXiv:1510.02428* (cit. on p. 5).
- 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).