Contents

Chapter 0

Draft to be distilled in 1 and 2

TODO: brief intro of chapter

Contents

0.1 Presentation of the domains

In this section, we recall or rigorously redefine notions related to tensors, neural networks and graphs. Vector spaces considered below are assumed to be finite-dimensional and over the field of real numbers $\mathbb{R}$.

0.1.1 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 is usually defined as a special class of multilinear functions (Bass, 1968; Williamson, 2015). As such, a mathematical tensor is entirely defined on the cartesian product of the canonical bases onto which its outputs can be represented by a multidimensional array. In that sense, both definitions rejoin on their representation, but the underlying objects are different. In particular, mathematical tensors enjoy a more intrinsic definition as they neither depend on their representation nor on a change of basis, whereas in our domain, they are confounded with their representation.
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 and any deep learning object can be later define rigorously.

Definition 0.1.1. 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 0.1.2. 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 $n_1^{(t)} \times n_2^{(t)} \times \cdots \times n_r^{(t)}$.

Definition 0.1.3. 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 0.1.1. This 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 is not a concern in our domain as we need not distinguish tensors from their representation.

Remark 0.1.2. For naming conveniency, 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 can be seen as a tensor space of rank 1. If there is no notion of rank defined, 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) to disambiguate with another notion of rank.

Definition 0.1.4. 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 $\mathbb{I} = \prod_{k=1}^{r} \{1, \ldots, n_k\}$ is called the *index space* of $\mathbb{T}$

Remark 0.1.3. When using an index i_k for an entry of a tensor t, we implicitly assume that $i_k \in \{1, 2, \cdots, n_k^{(t)}\}$ if nothing is specified.

Definition 0.1.5. 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 0.1.4. 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 0.1.6. 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:

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

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 0.1.7. 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 $g : \prod_{k=1}^{r} \{1, \dots, n_k\} \rightarrow \{1, \dots, n\}$ such that

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

An inverse operation is called a *de-flatten* operation.

Remark 0.1.5. **Row major ordering**
The choice of g determines in which order the indexing is made. g is reminescent of how data of multidimensional arrays or tensors are stored internally by programming languages. In most tensor manipulation languages, incrementing the memory adress (*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

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

Definition 0.1.8. 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.

Definition 0.1.9. 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 $\forall k \in \{1, 2, \ldots, s\}, n_{r_1-(s-k)}^{(1)} = n_k^{(2)}$, then the tensor contraction between $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$ is defined as:

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

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

Definition 0.1.10. 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 0.1.6. **Transformation law independency**

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

Remark 0.1.7. **Einstein summation convention**

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

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

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

Proposition 0.1.1. Matrix product equivalence
Using reshapings that groups all covariant indices into a single index and all contravariant indices into another single index, any tensor 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 as suggested in the remark, 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 0.1.11. 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 $\forall p \in \{1, 2, \dots, n\}, n_p^{(1)} \geq n_p^{(2)}$, is defined as:

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

Definition 0.1.12. 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 $\forall p \in \{1, 2, \dots, n\}, n_p^{(1)} \geq n_p^{(2)}$, is defined as:

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

Remark 0.1.8. Unformally, a strided convolution is defined as if it were a regular subsampling of a convolution. They match if $s = (1, 1, \dots, 1)$.

Definition 0.1.13. Pooling
Let a real-valued function f defined over any tensor space 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.

0.1.2 Neural Networks

0.1.2.1 Description

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

Definition 0.1.14. Neural network (simply connected)
Let F be a function such that I_f and O_f are vector or tensor spaces.
F is a *functional formulation* of a *simply connected neural network* if there are a series of linear or 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:

$$\begin{cases} \forall k \in \{1, 2, \ldots, L\}, f_k = h_k \circ g_k, \\ I_F = I_{f_1} \xrightarrow{f_1} O_{f_1} \cong I_{f_2} \xrightarrow{f_2} \ldots \xrightarrow{f_L} O_{f_L} = O_F, \\ F = f_L \circ ... \circ f_2 \circ f_1 \end{cases}$$

The couple (g_k, h_k) is called the *k-th layer* of the neural network. 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.

TODO: introduce what is their purpose ie classifiers, why is training, and make a little plan of what follows
TODO: remarks on universal approximators and refs, overfitting, generalization

Definition 0.1.15. Activation function
Let a layer (g, h). h is called the *activation function* of the layer. It is non-linear, derivable and univariate. Of common use for univariate functions is the functional notation $h(v)[i_1, i_2, \ldots, i_r] = h(v[i_1, i_2, \ldots, i_r])$.

Remark 0.1.9. **Example of activation functions**
TODO: blabla: refs activation functions

Definition 0.1.16. 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 $\mathcal{L}$.

Remark 0.1.10. **Bias**
Affine functions $\widetilde{g}$ can be written as a sum between a linear function g and a constant vector b which is called the *bias*. Its role is to augment the expressivity of the neural network's family of functions. For notational conveniency, we will omit the biases in the rest of this section and thus only consider linear functions.

Definition 0.1.17. 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.

Remark 0.1.11. **Biological inspiration**
A *(computational) neuron* is a computational unit that is biologically inspired. Each neuron should be 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 connexions between each successive layers. A neuron is illustrated on Figure 1.

placeholder

Figure 1: A neuron

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 0.1.18. Branching
A *binary branching operation* of a neural network is an operation between two activations, $x_{k_1} \bowtie x_{k_2}$, that outputs, subject to shape compatibility, either their addition, either their concatenation along a rank, or their concatenation as a list.
A *branching operation* of a neural network between n activations, $x_{k_1} \bowtie x_{k_2} \bowtie \cdots \bowtie x_{k_n}$, is a composition of binary branching operations, or is the identity function Id if $n = 1$.

Definition 0.1.19. Neural network (generic definition)
The set of *neural network functions* $\mathcal{N}$ is defined inductively as follow

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

Remark 0.1.12. **Examples**
TODO: blabla: residual connections, skip connections,branching layers

0.1.2.2 Training

Definition 0.1.20. Weights
Let consider the k-th layer of a neural networks. 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 $n_1^{(\theta_k)}$.

Remark 0.1.13. **Learning**
A *loss* function $\mathcal{L}$ penalizes the output $x_L = F(x)$ relatively to what can be expected. Gradient w.r.t. θ_k, denoted $\vec{\bigtriangledown}_{\theta_k}$, is used to update the weights via an optimization algorithm based on gradient descent and a learning rate α, that is:

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

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

TODO: examples of optimizations

Remark 0.1.14. **Linear complexity**
The complexity of computing the gradients is linear with the number of weights.

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

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

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

Obtaining,

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

where $J_{\text{wrt}}(.)$ are the respective jacobians which can be determined with the layer's expressions and the $\{x_k\}$; and $\vec{\bigtriangledown}_{x_L}$ can be determined using $\mathcal{L}$, R and x_L. $\square$

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).

Remark 0.1.15. **Back propagation**
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.

0.1.2.3 Example of layers

Definition 0.1.21. 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 0.1.22. 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 0.1.23. 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 0.1.24. 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 0.1.25. 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 0.1.24 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 0.1.16. 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.

Remark 0.1.17. 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, Agarwal, Barham, 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 0.1.26. 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 0.1.24, and g_{pad} is an operation that pads zeros to its inputs such that g preserves tensor shapes.

Remark 0.1.18. 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.* Remark 0.1.12. On the other hand, they increase memory and computational footprints.

Proposition 0.1.2. 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 0.1.11 that

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

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

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

Following Proposition 0.1.1, we reshape (10) as a matrix product. To reshape $y \mapsto Y$, we use the row major order bijections g_j as in (1) defined onto $\{(j_1, \dots, j_n), \forall t, 1 \leq j_t \leq n_t^{(y)}\}$. To reshape $x \mapsto X$, we use the same row major order bijection g_j, however defined on the indices that support non zero-padded values, so that zero-padded values are lost after reshaping. That is, we use a bijection g_i such that $g_i(i_1, i_2, \dots, i_n) = g_j(i_1 - o_1, i_2 - o_2, \dots, i_n - o_n)$ defined if and only if $\forall t, 1 + o_t \leq i_t \leq n_t^{(y)}$, where the $\{o_t\}$ are the starting

offsets of the non zero-padded values. $\widetilde{w} \mapsto W$ is reshaped by using g_j for its covariant indices, and g_i for its contravariant indices. The entries lost by using g_i do not matter because they would have been nullified by the resulting matrix product. We remark that W is exactly the block (p, q) of W_g (and not of $W_{g'}$). Now let's prove that it is a Toeplitz matrix.
Thanks to the linearity of the expression (1) of g_j, by denoting $i'_t = i_t - o_t$, we obtain

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

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

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

Hence, on a diagonal of W, $g_i - g_j$ remaining constant means that $W[g_i, g_j]$ also remains constants. So W is a Toeplitz matrix.
The converse is also true as we used invertible functions in the index spaces through the proof. □

Remark 0.1.19. 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.

Remark 0.1.20. Comparitively 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 datsets, their usage also breeds a significant boost in performance compared with dense layers (Krizhevsky, Sutskever, and Hinton, 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 0.1.27. Stride
A convolutional layer with *stride* is a convolutional layer that computes strided convolutions (with stride > 1) instead of convolutions.

Definition 0.1.28. 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.

Remark 0.1.21. **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 are the infomations captured by the weights of the layer.

TODO: below

0.1.2.4 Example of regularizations

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, Hinton, Krizhevsky, 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.

0.1.2.5 Example of architectures

TODO: rephrase

A multilayer perceptron (MLP) (Hornik, Stinchcombe, and White, 1989) is a neural network composed of only dense layers. A convolutional neural network (CNN) (LeCun, Bottou, Bengio, 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,

Stinchcombe, and White, 1989) via the chain rule and according to a chosen optimization algorithm (*e.g.* Bottou, 2010).

0.1.3 Graphs

TODO: check this subsection

A graph G is defined as a couple (V, E) where V represents the set of nodes and $E \subseteq \binom{V}{2}$ is the set of edges connecting these nodes.
TODO: Example of figure

We encounter 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, coined *connectivity graph*. It encodes how the information is propagated through a layer to another. See Section 0.1.3.1.
- A computation graph is used by deep learning frameworks to keep track of the dependencies between layers of a deep learning models, in order to compute forward and back-propagation. See Section 0.1.3.2.
- A graph can represent the underlying structure of an object (often a vector), whose nodes represent its features. See Section 0.1.3.3.
- Datasets can also be graph-structured, where the nodes represent the objects of the dataset. See Section 0.1.3.4.

0.1.3.1 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 $N = \{u_1, u_2, \ldots, u_n\}$, and m the size of the set of output neurons $M = \{v_1, v_2, \ldots, v_m\}$. This layer implements the equation $y = \Theta x$ where Θ is a $n \times m$ matrix.

Definition 0.1.29. 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 2 depicts some examples.

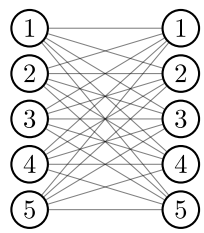

Figure 2: Examples

TODO: Figure 2. It's just a placeholder right now

Connectivity graphs also allow to graphically modelize how weights are tied in a neural layer. Let's suppose the $\Theta_i j$ are taking their values only into the finite set $K = \{w_1, w_2, \ldots, w_\kappa\}$ of size κ, which we will refer to as the *kernel* of *weights*. Then we can define a labelling of the edges $s : E \to K$. s is called the *weight sharing scheme* of the layer. This layer can then be formulated as $\forall v \in M, y_v = \sum_{u \in N, (u,v) \in E} w_{s(u,v)} x_u$. Figure 3 depicts the connectivity graph of a 1-d convolution layer and its weight sharing scheme.

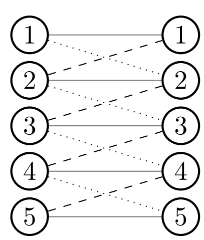

Figure 3: Depiction of a 1D-convolutional layer and its weight sharing scheme.

TODO: Add weight sharing scheme in Figure 3

0.1.3.2 Computation graph

0.1.3.3 Underlying graph structure

0.1.3.4 Graph-structured dataset

transductive vs inductive

0.1.4 Special classes of graphs

0.1.4.1 Grid graphs

0.1.4.2 Spatial graphs

0.1.4.3 Projections of spatial graphs

0.2 Subject disambiguation

TODO: Rework 1.1

0.2.1 Naming conventions

0.2.1.1 Basic notions

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

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

0.2.1.2 Graphs and graph signals

TODO: more defs on grid graphs and other graphs
A *graph* $G = (V, E)$ is defined as a set of nodes V, and a set of edges $E \subseteq \binom{V}{2}$.
The words *node* and *vertex* will be used equivalently, but we will rather use the first.
A *graph signal*, or *graph-structured signal* is a signal defined on the nodes of a graph, for which the underlying structure is the graph itself. A *node signal*

is a signal defined on a node, in which case it is a *node embedding* in a vector space.
Although this is rarely seen, a signal can also be defined on the edges of a graph, or on an edge. We then coin it respectively *dual graph signal*, or *edge signal / edge embedding.*
Graph-structured data can refer to any of these type of signals.

0.2.1.3 Data and datasets

A dataset of signals is said to be *static* if all its signals share the same underlying structure, it is said to be *non-static* otherwise.
For image datasets, being non-static would mean that the dataset contains images of different sizes or different scales. For graph signal datasets, it would mean thats the underlying graph structures of the signals are different.
The point in specifying that objects of a dataset of a machine learning task are signals is that we can hope to leverage their underlying structure.
TODO: figure

0.2.2 Disambiguation of the subject

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

0.2.2.1 Irregularly structured data

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

Definition 0.2.1. Structure
Let $s : D \to F$ be a signal defined over a finite domain.
An *underlying structure* of the signal s is a graph G that has the domain of s for nodes.

A dataset is said to be *structured*, if its objects can be modelized as signals with an underlying structure.
It is said to be *static* if all its objects share the same underlying structure, and *non-static* otherwise.

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

Definition 0.2.2. Regularity
An underlying structure is said to be *regular*, if it is a regular grid graph. It is said to be *irregular* otherwise.
A dataset is said to be *regularly structured*, if the underlying structures of its objects are regular. It is said to be *irregularly structured* otherwise.

TODO: examples

0.2.2.2 Unstructured data

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

0.2.3 Datasets

0.2.4 Tasks

0.2.5 Goals

0.2.6 Invariance

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

0.2.7 Methods

Contents

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

Temptative titles

- Learning propagational representations of irregular and unstructured data
- Learning representations of unstructured or irregularly structured datasets
- Propagational learning of unstructured or irregularly structured datasets
- Learning tensorial representation of irregular and unstructured data
- Tensorial representation of propagation in deep learning for irregular and unstructured dataset
- Structural representation learning for irregular or unstructured data
- Word for both "irregularly structured" + "unstructured" = ? (maybe "unorthodox" ?)
- Unorthdox deep learning
- ...

- Deep learning of unstructured or irregularly structured datasets
- Deep learning models for data without a regular structure
- On structures in deep learning

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. 17).

Bass, Jean (1968). "Cours de mathématiques". In: (cit. on p. 7).

Bottou, Léon (2010). "Large-scale machine learning with stochastic gradient descent". In: *Proceedings of COMPSTAT'2010*. Springer, pp. 177–186 (cit. on p. 21).

Hackbusch, Wolfgang (2012). *Tensor spaces and numerical tensor calculus*. Vol. 42. Springer Science & Business Media (cit. on p. 8).

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366 (cit. on p. 20).

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 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 p. 19).

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. 20).

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. 20).

Williamson, S Gill (2015). "Tensor spaces-the basics". In: *arXiv preprint arXiv:1510.02428* (cit. on p. 7).