

Chapter 1

First chapter

1.1 Definitions and disambiguations

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} .

1.1.1 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 linear 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. 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 multidimensional array representation nor on a change of basis.

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 linear space and any deep learning object can be later define rigorously.

Definition 1.1.1. Tensor space

We define a *tensor space* \mathbb{T} of rank r as a cartesian product of r vector spaces under the coordinate-wise sum and the mono-linear outer product.

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.

Hence, a tensor space is a linear space. Note that for naming convenience, we distinguish between the terms *linear space* and *vector space*. That is, we 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 term *dimension* (that is, for a tensor space it is equal to $\prod_{k=1}^r n_k$) and the term *rank* (equal to r).

Definition 1.1.2. 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 1.1.3. Indexing

An *entry* of a tensor t is a scalar denoted $t[i_1, i_2, \dots, i_r]$.

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 $\{l_p\}$ and the $\{u_p\}$ are the lower and upper bounds of the indices used by the entries that compose t' .

When using an index i_k for an entry of a tensor t , we implicitly assume that $i_k \in \{1, 2, \dots, n_k^{(t)}\}$ if nothing is precised. For subtensor indexings, 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 $n_p^{(t)}$.

Definition 1.1.4. 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 $s : \mathbb{T} = \prod_{k=1}^r \mathbb{V}_k \rightarrow \prod_{k=1}^{r-s} \mathbb{V}_k$, such that:

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

where $:=$ means that values 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 *sliced subspace*.

Definition 1.1.5. 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 char-

acterized 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 1.1.1. 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 increment only i_r if $i_r < n_r$ (and then ranks are ordered in reverse lexicographic order to decide what index is also incremented). This is called *row major ordering*, as opposed to *column major ordering*. That is, in row major, g is defined as

$$g(i_1, i_2, \dots, i_r) = \sum_{p=1}^r \left(\prod_{k=p+1}^r n_k \right) i_p \quad (1.1)$$

Definition 1.1.6. Reshaping

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

Definition 1.1.7. 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 $\forall k \in \{1, 2, \dots, s\}, n_{r_1-(s-k)}^{(1)} = n_k^{(2)}$, then the tensor contraction between $t_1 \in \mathbb{T}_1$ and $t_2 \in \mathbb{T}_2$ is defined as:

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

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 1.1.8. 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 1.1.2. Transformation law independency

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

Remark 1.1.3. 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:

$$t_{1_{i_1^{(1)} \dots i_{r_1-s}^{(1)}}}^{k_1 \dots k_s} t_{2_{k_1 \dots k_s}}^{i_{s+1}^{(2)} \dots i_{r_2}^{(2)}} = t_{3_{i_1^{(1)} \dots i_{r_1-s}^{(1)}}}^{i_{s+1}^{(2)} \dots i_{r_2}^{(2)}} \quad (1.2)$$

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.

Remark 1.1.4. Matrix product equivalence

Using reshaping 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 (1.2), with the reshaping $t_1 \mapsto T_1$, $t_2 \mapsto T_2$ 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 (1.1). \square

Definition 1.1.9. 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:

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

Definition 1.1.10. Pooling

TODO: Use indexing, maybe add stride in this subsection rather than in the next

1.1.2 Neural Networks**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 1.1.11. 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 $(g_k)_{k=1,2,\dots,L}$ and a series of non-linear derivable univariate functions $(h_k)_{k=1,2,\dots,L}$ such that:

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

The couple (g_k, h_k) is called the k -th *layer* of the neural network. For $x \in I_f$, we denote by $x_k = f_k \circ \dots \circ f_2 \circ f_1(x)$ 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 1.1.12. 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 $h(v)[i_1, i_2, \dots, i_r] = h(v[i_1, i_2, \dots, i_r])$.

Remark 1.1.5. Example of activation functions

TODO: blabla: refs activation functions

Definition 1.1.13. 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 1.1.6. Bias

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*. 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 1.1.14. 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 1.1.7. 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 1.1.

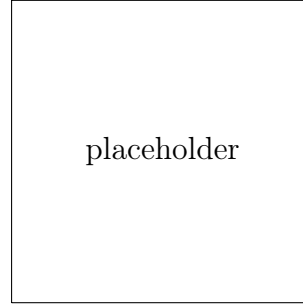


Figure 1.1: A neuron

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 1.1.15. 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, $x_{k_1} \bowtie x_{k_2} \bowtie \dots \bowtie x_{k_n}$, is a composition of binary branching operations, or is the identity function Id if $n = 1$.

Definition 1.1.16. 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:
 $f_1, f_2, \dots, f_n \in \mathcal{N} \Rightarrow f_1 \bowtie f_2 \bowtie \dots \bowtie f_n \in \mathcal{N}$

TODO: blabla: residual connections, skip connections, branching layers

Training

Definition 1.1.17. 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 1.1.8. 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 $\vec{\nabla}_{\theta_k}$, is used to update the weights via an optimization algorithm based on gradient descent and a learning rate α , that is:

$$\theta_k^{(\text{new})} = \theta_k^{(\text{old})} - \alpha \cdot \vec{\nabla}_{\theta_k} \left(\mathcal{L} \left(x_L, \theta_k^{(\text{old})} \right) + R \left(\theta_k^{(\text{old})} \right) \right) \quad (1.3)$$

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 1.1.9. 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, $\vec{\nabla}_{\theta_k}$ 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:

$$\vec{\nabla}_{\theta_k} = J_{\theta_k}(x_k) \vec{\nabla}_{x_k} \quad (1.4)$$

$$\begin{aligned} \vec{\nabla}_{x_k} &= J_{x_k}(x_{k+1}) \vec{\nabla}_{x_{k+1}} \\ \vec{\nabla}_{x_{k+1}} &= J_{x_{k+1}}(x_{k+2}) \vec{\nabla}_{x_{k+2}} \\ &\dots \end{aligned} \quad (1.5)$$

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

Obtaining,

$$\vec{\nabla}_{\theta_k} = J_{\theta_k}(x_k) \left(\prod_{p=k}^{L-1} J_{x_p}(x_{p+1}) \right) \vec{\nabla}_{x_L} \quad (1.6)$$

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 . \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 1.1.10. Back propagation

We can remark that (1.5) rewrites as

$$\begin{aligned} \vec{\nabla}_{x_k} &= J_{x_k}(x_{k+1}) \vec{\nabla}_{x_{k+1}} \\ &= J_{x'_k}(h(x'_k)) J_{x_k}(W_k x_k) \vec{\nabla}_{x_{k+1}} \end{aligned} \quad (1.7)$$

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

$$J_{x'_k}(h(x'_k))[i, j] = \delta_i^j h'(x'_k[i]) \quad (1.8)$$

$$\begin{aligned} J_{x'_k}(h(x'_k)) &= I h'(x'_k) \\ J_{x_k}(W_k x_k) &= W_k^T \end{aligned} \quad (1.9)$$

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 \tilde{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.

Example of layers**Definition 1.1.18. 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 1.1.19. 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 1.1.20. 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 1.1.21. 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. The slices $g(x)_q$ are typically called *feature maps*.

Remark 1.1.11. Geometric shape **TODO: blabla**

Definition 1.1.22. Padding

A layer (g, h) with padding is such that $\exists (g_{\text{pad}}, g'), g = g_{\text{pad}} \circ g'$ where g_{pad} is a padding operation.

A convolutional layer with padding $(g_{\text{pad}} \circ g', h)$ is such that g_{pad} **TODO: rewrite**

TODO: blabla padding

Proposition 1.1.1. Connectivity matrix of a convolution with padding

A convolutional layer with padding (g, h) is equivalently defined as W_g being a $n_{n+1}^{(I_g)} \times n_{n+1}^{(O_g)}$ 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 1.1.9 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 (1.2) and permuting indices, we recognize the following tensor contraction

$$y_{j_1 \dots j_n} = x_{i_1 \dots i_n} \tilde{w}^{i_1 \dots i_n}_{j_1 \dots j_n} \quad (1.10)$$

Following remark 1.1.4, we reshape (1.10) as a matrix product. To reshape $y \mapsto Y$, we use the row major order bijections g_j as in (1.1) 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 (1.1) of g_j , by denoting $i'_t = i_t - o_t$, we obtain

$$g_i(i_1, i_2, \dots, i_n) - g_j(j_1, j_2, \dots, j_n) = g_j(i'_1 - j_1, i'_2 - j_2, \dots, i'_n - j_n) \quad (1.11)$$

To simplify the notations, let's drop the arguments of g_i and g_j . By bijectivity of g_j , (1.11) tells us that $g_i - g_j$ remains constant if and only if $i'_t - j_t$ remains constant for all t . Recall that

$$W[g_i, g_j] = \begin{cases} w[j_1 + n_1^{(w)} - i'_1, \dots, j_n + n_n^{(w)} - i'_n] & \text{if } \forall t, 0 \leq i'_t - j_t \leq n_t^{(w)} - 1 \\ 0 & \text{otherwise} \end{cases} \quad (1.12)$$

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

TODO: remark when no padding

TODO: below

Definition 1.1.23. Convolutional layer with stride

Let g the linear part of a convolution layer with *stride* $s_p > 1$ along the p -th rank, and \tilde{g} the same linear part if it was a regular convolutional layer as defined above. Then g is defined as

$$\forall i_p \in \{1, \lfloor \frac{n_p^{(g(x))}}{s_p} \rfloor\}, \forall x \in I_g, g(x)_{i_p} = \tilde{g}(x)_{s_p i_p}$$

where i_p and $s_p i_p$ index slices along the p -th rank.

TODO: below

Definition 1.1.24. Pooling

A layer with *pooling* (g, h) is such that $g = g_1 \circ g_2$, where (g_1, h) is a layer and g_2 is a pooling operation.

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 [?]. 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.

Example of architectures

TODO: rephrase

A multilayer perceptron (MLP) [?] is a neural network composed of only dense layers. A convolutional neural network (CNN) [?] 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 [?] via the chain rule and according to a chosen optimization algorithm (e.g. [?]).

1.1.3 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 1.1.3.
- 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 1.1.3.
- A graph can represent the underlying structure of an object (often a vector), whose nodes represent its features. See section 1.1.3.
- Datasets can also be graph-structured, where the nodes represent the objects of the dataset. See section 1.1.3.

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 1.1.25. 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 1.2 depicts some examples.

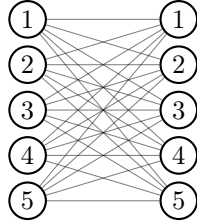


Figure 1.2: Examples

TODO: Figure 1.2. 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 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 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 1.3 depicts the connectivity graph

of a 1-d convolution layer and its weight sharing scheme.

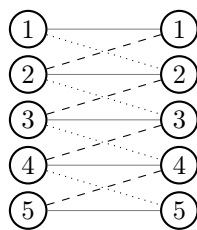


Figure 1.3: Depiction of a 1D-convolutional layer and its weight sharing scheme.

TODO: Add weight sharing scheme in Figure 1.3

Computation graph

Underlying graph structure

Graph-structured dataset

transductive vs inductive

1.1.4 Geometric grids

1.1.5 Grid graphs

1.1.6 Spatial graphs

1.1.7 Projections of spatial graphs