

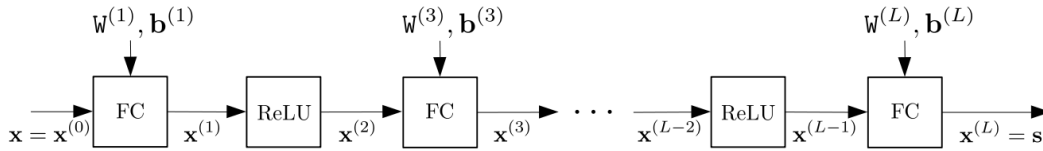
# TD : Implémentation d'un réseau de neurones de type Perceptron multicouche pour un problème de classification

Guillaume Bourmaud

28/01/24

## 1 Introduction

Un réseau de neurones de type Perceptron multicouche (MLP) est composé d'une succession de couches, alternant une transformation affine (FC) et une fonction non-linéaire (historiquement la fonction sigmoïde mais ici nous utiliserons la fonction ReLU :  $z = \max(0, x)$ ). Un tel réseau est illustré graphiquement ci-après :



Remarquons que selon les notations utilisées dans le schéma précédent, le nombre de couches  $L$  est nécessairement impair et supérieur ou égal à 3 (car les couches d'entrée et de sortie sont des transformations affines). Ainsi un MLP s'écrit comme une composition de fonctions :

$$\mathbf{s} = \text{MLP} \left( \mathbf{x}; \left\{ \mathbf{w}^{(2j-1)}, \mathbf{b}^{(2j-1)} \right\}_{j=1, \dots, \lceil L/2 \rceil} \right) \quad (1)$$

$$= \text{FC} \left( \text{ReLU} \left( \dots \text{FC} \left( \text{ReLU} \left( \text{FC} \left( \mathbf{x}; \mathbf{w}^{(1)}, \mathbf{b}^{(1)} \right) \right) \dots \right); \mathbf{w}^{(3)}, \mathbf{b}^{(3)} \right) \dots; \mathbf{w}^{(L)}, \mathbf{b}^{(L)} \right). \quad (2)$$

Nous considérons le cas où les vecteurs  $(\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(L-1)})$  sont tous de taille  $H$  et sont des vecteurs ligne, donc de taille  $1 \times H$ . Le MLP que nous considérons a donc deux hyper-paramètres :  $L$  (le nombre de couches) et  $H$ . Nous nous intéressons ici à un problème de classification à  $C$  classes, donc  $\mathbf{x}^{(L)} = \mathbf{s}$  est de taille  $1 \times C$ . L'espace de départ est de dimension  $D$  donc  $\mathbf{x} = \mathbf{x}^{(0)}$  est de taille  $1 \times D$ . La fonction de coût utilisée est l'entropie croisée Multinoulli :

$$l(y, \mathbf{s}) = -\ln(\mathbf{p}_y) \quad (3)$$

où  $\mathbf{p} = \text{softmax}(\mathbf{s})$  est un vecteur de probabilités (scores positifs qui se somment à 1) de taille  $1 \times C$  et  $y \in \{0, \dots, C-1\}$  est l'étiquette (donc un scalaire  $1 \times 1$ ). Par exemple, si  $y = 2$ , alors la notation  $\mathbf{p}_y$  correspond à récupérer le 3ème élément du vecteur  $\mathbf{p}$ . Minimiser la fonction de coût revient donc à maximiser le score de probabilité  $\mathbf{p}_y$ . La fonction softmax est définie de la manière suivante :

$$\mathbf{p}_i = \frac{\exp(\mathbf{s}_i)}{\sum_{j=0}^{C-1} \exp(\mathbf{s}_j)}. \quad (4)$$

Les paramètres du réseau seront optimisés en minimisant la fonction de coût suivante sur la base d'apprentissage  $\{\mathbf{x}_{[i]}, y_{[i]}\}_{i=0 \dots N-1}$  :

$$\arg \min_{\{\mathbf{w}^{(2j-1)}, \mathbf{b}^{(2j-1)}\}_{j=1, \dots, \lceil L/2 \rceil}} \frac{1}{N} \sum_{i=0}^{N-1} l \left( y_{[i]}, \text{MLP} \left( \mathbf{x}_{[i]}; \left\{ \mathbf{w}^{(2j-1)}, \mathbf{b}^{(2j-1)} \right\}_{j=1, \dots, \lceil L/2 \rceil} \right) \right) \quad (5)$$

Nous utiliserons pour cela une méthode de descente de gradient. Comme vu en cours, le gradient, c'est-à-dire la dérivée de la fonction de coût par rapport aux paramètres du réseau, est obtenu en utilisant le théorème de dérivation des fonctions composées.

## 2 Théorème de dérivation d'une fonction composée

Prenons le cas d'une fonction composée de deux fonctions ( $f : \mathbb{R}^D \rightarrow \mathbb{R}^H$  et  $g : \mathbb{R}^H \rightarrow \mathbb{R}$ ) :

$$a = g(f(\mathbf{x})). \quad (6)$$

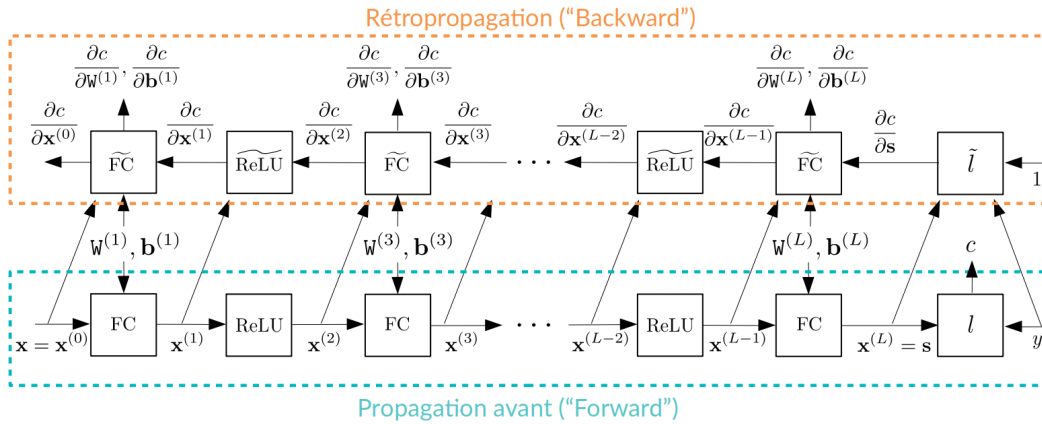
En définissant  $\mathbf{z} = f(\mathbf{x})$ , le théorème de dérivation d'une fonction composée nous indique que la dérivée de  $a \in \mathbb{R}$  par rapport au  $i$ -ème élément du vecteur  $\mathbf{x}$  s'exprime de la manière suivante :

$$\frac{\partial a}{\partial \mathbf{x}_i} = \sum_{j=0}^{H-1} \frac{\partial a}{\partial \mathbf{z}_j} \bigg|_{\mathbf{z}=f(\mathbf{x})} \frac{\partial \mathbf{z}_j}{\partial \mathbf{x}_i}. \quad (7)$$

Ainsi nous pouvons introduire la notation suivante :

$$\frac{\partial a}{\partial \mathbf{x}} = \tilde{f} \left( \frac{\partial a}{\partial \mathbf{z}} \bigg|_{\mathbf{z}=f(\mathbf{x})}, \mathbf{x} \right), \quad (8)$$

où  $\tilde{f}$  est la fonction qui implémente l'équation (7) (c'est-à-dire qui « rétropropage »  $\frac{\partial a}{\partial \mathbf{z}} \big|_{\mathbf{z}=f(\mathbf{x})}$  pour obtenir  $\frac{\partial a}{\partial \mathbf{x}}$ ),  $\frac{\partial a}{\partial \mathbf{x}} = \left[ \frac{\partial a}{\partial \mathbf{x}_0}, \frac{\partial a}{\partial \mathbf{x}_1}, \dots, \frac{\partial a}{\partial \mathbf{x}_{D-1}} \right]$  et  $\frac{\partial a}{\partial \mathbf{z}} \big|_{\mathbf{z}=f(\mathbf{x})} = \left[ \frac{\partial a}{\partial \mathbf{z}_0}, \frac{\partial a}{\partial \mathbf{z}_1}, \dots, \frac{\partial a}{\partial \mathbf{z}_{H-1}} \right]$ . En appliquant ce théorème à un MLP dans le but de calculer la dérivée de la fonction de coût par rapport aux paramètres du MLP, nous obtenons l'étape de rétropropagation du gradient, sous la forme d'un **graphe de calcul**, illustrée dans la figure ci-après.



**Travail : écrire sur le graphe de calcul ci-dessus la taille de chacune des variables puis calculer le nombre de paramètres du MLP**

Ainsi, pour mettre en œuvre une méthode de descente de gradient dans le but d'optimiser les paramètres d'un MLP, nous aurons uniquement besoin de connaître les expressions des fonctions  $\tilde{l}(1, \mathbf{s})$ ,  $\widetilde{\text{FC}} \left( \frac{\partial l}{\partial \mathbf{x}^{(i)}} \bigg|_{\mathbf{x}^{(i)} = \text{FC}(\mathbf{x}^{(i-1)})}, \mathbf{x}^{(i-1)}, W^{(i)}, \mathbf{b}^{(i)} \right)$  et  $\widetilde{\text{ReLU}} \left( \frac{\partial l}{\partial \mathbf{x}^{(i)}} \bigg|_{\mathbf{x}^{(i)} = \text{ReLU}(\mathbf{x}^{(i-1)})}, \mathbf{x}^{(i-1)} \right)$ . C'est le travail qui sera réalisé dans la suite de ce TD. Les expressions obtenues seront codées ultérieurement lors d'un TP en Python utilisant la bibliothèque Numpy.

## 3 Obtention des expressions de $\widetilde{\text{FC}}$ , $\widetilde{\text{ReLU}}$ et $\tilde{l}$

### 3.1 Dérivée transformation affine : $\widetilde{\text{FC}}$

En utilisant les notations précédemment introduites, une transformation affine s'écrit :

$$\mathbf{z}_l = \sum_{j=0}^{D-1} \mathbf{x}_j W_{jl} + \mathbf{b}_l \quad (9)$$

$$\text{où } \underbrace{\mathbf{z}}_{1 \times H} = \begin{bmatrix} \mathbf{z}_0 & \mathbf{z}_1 & \dots & \mathbf{z}_{H-1} \end{bmatrix}, \underbrace{\mathbf{W}}_{D \times H} = \begin{bmatrix} \mathbf{W}_{00} & \mathbf{W}_{01} & \dots & \mathbf{W}_{0(H-1)} \\ \mathbf{W}_{10} & \mathbf{W}_{11} & \dots & \mathbf{W}_{1(H-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{W}_{(D-1)0} & \mathbf{W}_{(D-1)1} & \dots & \mathbf{W}_{(D-1)(H-1)} \end{bmatrix} \text{ et } \underbrace{\mathbf{b}}_{1 \times H} = \begin{bmatrix} \mathbf{b}_0 \\ \mathbf{b}_1 \\ \vdots \\ \mathbf{b}_{(H-1)} \end{bmatrix}^\top.$$

En utilisant le produit matrice/vecteur, l'équation devient :

$$\mathbf{z} = \text{FC}(\mathbf{x}, \mathbf{W}, \mathbf{b}) = \mathbf{x}\mathbf{W} + \mathbf{b}. \quad (10)$$

La sortie  $\mathbf{Z}$  de cette transformation affine est passée dans une fonction  $g$  dont la sortie  $c$  est scalaire (car en pratique on dérive toujours par rapport à un coût qui par définition est scalaire) :

$$c = g(\mathbf{z}). \quad (11)$$

En supposant le gradient  $\underbrace{\partial c / \partial \mathbf{z}}_{1 \times H} = \begin{bmatrix} \partial c / \partial \mathbf{z}_0 & \partial c / \partial \mathbf{z}_1 & \dots & \partial c / \partial \mathbf{z}_{H-1} \end{bmatrix}$  **connu**, l'objectif est de calculer la dérivée de  $c$  par rapport à  $\mathbf{x}$  et aux paramètres  $\mathbf{W}$  et  $\mathbf{b}$ .

**Travail : dessiner le graphe de calcul correspondant aux deux équations précédentes, en faisant apparaître  $c, g, \mathbf{z}, \text{FC}, \mathbf{x}, \mathbf{W}, \mathbf{b}, \tilde{g}, \partial c / \partial \mathbf{z}, \widetilde{\text{FC}}, \partial c / \partial \mathbf{x}, \partial c / \partial \mathbf{W}$  et  $\partial c / \partial \mathbf{b}$ .**

L'objectif de cette partie est d'obtenir des expressions permettant d'implémenter la fonction  $\widetilde{\text{FC}}$ . Il faut donc calculer les expressions de  $\partial c / \partial \mathbf{x}$ ,  $\partial c / \partial \mathbf{W}$  et  $\partial c / \partial \mathbf{b}$  en les simplifiant au maximum pour faire apparaître des opérations permettant une implémentation efficace en Python (par exemple des produits de matrices).

### 3.1.1 Calcul de $\partial c / \partial \mathbf{x}$

Le théorème de dérivation d'une fonction composée nous dit que :

$$\partial c / \partial \mathbf{x}_i = \sum_{l=0}^{H-1} \partial c / \partial \mathbf{z}_l \cdot \partial \mathbf{z}_l / \partial \mathbf{x}_i \quad (12)$$

**Travail : expliciter  $\partial \mathbf{z}_l / \partial \mathbf{x}_i$  puis montrer que  $\underbrace{\partial c / \partial \mathbf{x}}_{1 \times D} = \underbrace{\partial c / \partial \mathbf{z}}_{1 \times H} \underbrace{\mathbf{W}^\top}_{H \times D}$**

### 3.1.2 Calcul de $\partial c / \partial \mathbf{W}$

Le théorème de dérivation d'une fonction composée nous dit que :

$$\partial c / \partial \mathbf{W}_{mn} = \sum_{l=0}^{H-1} \partial c / \partial \mathbf{z}_l \cdot \partial \mathbf{z}_l / \partial \mathbf{W}_{mn} \quad (13)$$

**Travail :**

— expliciter  $\partial \mathbf{z}_l / \partial \mathbf{W}_{mn}$

— montrer que l'équation (13) se simplifie en  $\partial c / \partial \mathbf{W}_{mn} = \partial c / \partial \mathbf{z}_n \cdot \mathbf{x}_m$

— en posant  $\underbrace{\partial c / \partial \mathbf{W}}_{D \times H} = \begin{bmatrix} \partial c / \partial \mathbf{W}_{00} & \partial c / \partial \mathbf{W}_{01} & \dots & \partial c / \partial \mathbf{W}_{0(H-1)} \\ \partial c / \partial \mathbf{W}_{10} & \partial c / \partial \mathbf{W}_{11} & \dots & \partial c / \partial \mathbf{W}_{1(H-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \partial c / \partial \mathbf{W}_{(D-1)0} & \partial c / \partial \mathbf{W}_{(D-1)1} & \dots & \partial c / \partial \mathbf{W}_{(D-1)(H-1)} \end{bmatrix}$ , **montrer que  $\underbrace{\partial c / \partial \mathbf{W}}_{D \times H} = \underbrace{\mathbf{x}^\top}_{D \times 1} \underbrace{\partial c / \partial \mathbf{z}}_{1 \times H}$**

### 3.1.3 Calcul de $\partial c / \partial \mathbf{b}$

**Travail : montrer que  $\underbrace{\partial c / \partial \mathbf{b}}_{1 \times H} = \underbrace{\partial c / \partial \mathbf{z}}_{1 \times H}$**

### 3.2 Dérivée ReLU : $\widetilde{\text{ReLU}}$

De manière générale, en utilisant les notations précédemment introduites, la fonction ReLU peut s'écrire :

$$\mathbf{z}_l = \max(0, \mathbf{x}_l) \quad (14)$$

La sortie  $\mathbf{Z}$  de cette fonction d'activation ReLU est passée dans une fonction  $g$  dont la sortie  $c$  est scalaire (car en pratique on dérive toujours par rapport à un coût qui par définition est scalaire) :

$$c = g(\mathbf{z}) \quad (15)$$

En supposant le gradient  $\underbrace{\partial c / \partial \mathbf{z}}_{1 \times H} = \begin{bmatrix} \partial c / \partial \mathbf{z}_0 & \partial c / \partial \mathbf{z}_1 & \dots & \partial c / \partial \mathbf{z}_{H-1} \end{bmatrix}$  connu, l'objectif est de calculer la dérivée de  $c$  par rapport aux entrées  $\mathbf{x}$ .

**Travail : dessiner le graphe de calcul correspondant aux deux équations précédentes, en faisant apparaître  $c$ ,  $g$ ,  $\mathbf{z}$ ,  $\widetilde{\text{ReLU}}$ ,  $\mathbf{x}$ ,  $\tilde{g}$ ,  $\partial c / \partial \mathbf{z}$ ,  $\widetilde{\text{ReLU}}$  et  $\partial c / \partial \mathbf{x}$ .**

**Travail :**

- montrer que  $\partial c / \partial \mathbf{x}_k = \begin{cases} \partial c / \partial \mathbf{z}_k & \text{si } \mathbf{x}_k > 0 \\ 0 & \text{si } \mathbf{x}_k < 0 \end{cases}$
- la dérivée est-elle définie en 0 ?

### 3.3 Dérivée de la fonction de coût : $\tilde{l}$

De manière générale, en utilisant les notations précédemment introduites, la fonction de coût  $l$  peut s'écrire :

$$c = -\ln \left( \frac{\exp(\mathbf{s}_y)}{\sum_{j=0}^{C-1} \exp(\mathbf{s}_j)} \right) \quad (16)$$

**Travail : dessiner le graphe de calcul correspondant à l'équation précédente, en faisant apparaître  $c$ ,  $l$ ,  $\mathbf{s}$ ,  $\tilde{l}$ ,  $\partial c / \partial \mathbf{s}$  et  $y$ .**

**Travail :**

- montrer que  $\partial c / \partial \mathbf{s}_k = \begin{cases} \mathbf{p}_k - 1 & \text{si } k = y \\ \mathbf{p}_k & \text{sinon} \end{cases}$
- interpréter ce résultat (Rappel : l'objectif était d'avoir une fonction de coût dont l'opposé du gradient permettrait d'augmenter le score de l'étiquette vérité terrain et de baisser les autres scores).

## 4 Parallélisation des calculs

Dans le but de paralléliser les calculs des gradients, et ainsi d'obtenir une implémentation en Python efficace, nous considérons le cas où les  $N$  vecteurs de taille  $D$   $\{\mathbf{x}_{[i]}\}_{i=0 \dots N-1}$  de la base d'apprentissage sont organisés en une matrice (en deep learning, on parle de « tenseur ») :

$$\underbrace{\mathbf{X}}_{N \times D} = \begin{bmatrix} \mathbf{x}_{00} & \mathbf{x}_{01} & \dots & \mathbf{x}_{0(D-1)} \\ \mathbf{x}_{10} & \mathbf{x}_{11} & \dots & \mathbf{x}_{1(D-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{x}_{(N-1)0} & \mathbf{x}_{(N-1)1} & \dots & \mathbf{x}_{(N-1)(D-1)} \end{bmatrix}. \quad (17)$$

Les étiquettes  $\{y_{[i]}\}_{i=0 \dots N-1}$  sont organisées sous la forme d'un vecteur :

$$\mathbf{y} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{(N-1)} \end{bmatrix}^\top \quad (18)$$

Ainsi le MLP s'appliquera directement sur ce tenseur  $\mathbf{X}$  et produira un tenseur de scores  $\mathbf{S}$  :

$$\mathbf{S} = \text{MLP}(\mathbf{X}; \boldsymbol{\theta}) = \text{FC}\left(\text{ReLU}\left(\dots \text{FC}\left(\text{ReLU}\left(\text{FC}\left(\mathbf{X}; \mathbf{W}^{(1)}, \mathbf{b}^{(1)}\right)\right); \mathbf{W}^{(3)}, \mathbf{b}^{(3)}\right) \dots\right); \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\right) \quad (19)$$

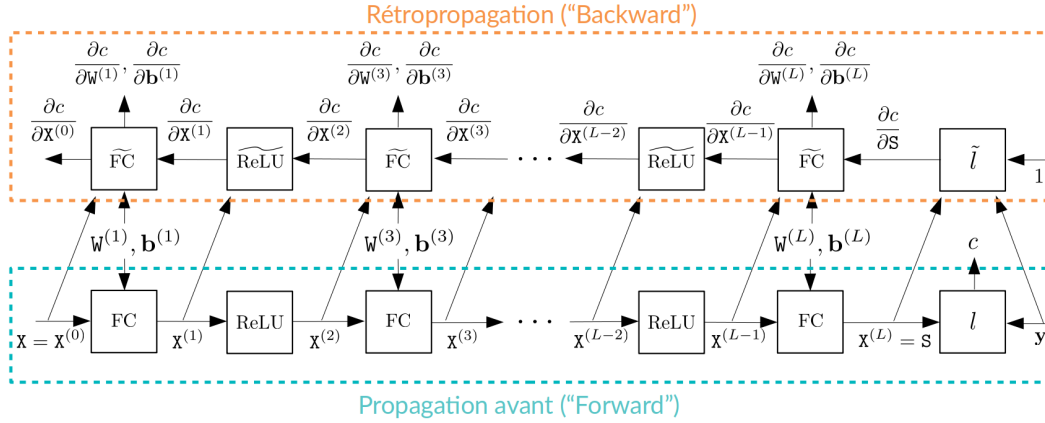
où

$$\underbrace{\mathbf{S}}_{N \times C} = \begin{bmatrix} \mathbf{S}_{00} & \mathbf{S}_{01} & \dots & \mathbf{S}_{0(C-1)} \\ \mathbf{S}_{10} & \mathbf{S}_{11} & \dots & \mathbf{S}_{1(C-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{S}_{(N-1)0} & \mathbf{S}_{(N-1)1} & \dots & \mathbf{S}_{(N-1)(C-1)} \end{bmatrix}. \quad (20)$$

Après softmax, les vecteurs de probabilité peuvent également être rangés dans une matrice :

$$\underbrace{\mathbf{P}}_{N \times C} = \begin{bmatrix} \mathbf{P}_{00} & \mathbf{P}_{01} & \dots & \mathbf{P}_{0(C-1)} \\ \mathbf{P}_{10} & \mathbf{P}_{11} & \dots & \mathbf{P}_{1(C-1)} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{P}_{(N-1)0} & \mathbf{P}_{(N-1)1} & \dots & \mathbf{P}_{(N-1)(C-1)} \end{bmatrix}. \quad (21)$$

Par définition, les matrices  $(\mathbf{X}^{(1)}, \dots, \mathbf{X}^{(L-1)})$  sont donc de taille  $N \times H$ .



**Travail : écrire sur le graphe de calcul ci-dessus la taille de chacune des variables**

Les expressions précédemment obtenues qui s'appliquent à un seul vecteur  $\mathbf{x}$  se généralisent trivialement au cas d'un tenseur  $\mathbf{X}$ . Ainsi la transformation affine se réécrit

$$\mathbf{Z} = \text{FC}(\mathbf{X}, \mathbf{W}, \mathbf{b}) = \mathbf{X}\mathbf{W} + \mathbf{1}_N \mathbf{b}, \quad (22)$$

où  $\underbrace{\mathbf{1}_N}_{N \times 1} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$ , et les 3 rétropropagations de  $\widetilde{\text{FC}}$  se réécrivent :

$$\underbrace{\frac{\partial c}{\partial \mathbf{X}}}_{N \times D} = \underbrace{\frac{\partial c}{\partial \mathbf{Z}}}_{N \times H} \underbrace{\mathbf{W}^\top}_{H \times D}, \quad (23)$$

qui pourra être codée en Python par un simple appel à une fonction réalisant un produit matriciel (@ en Numpy),

$$\underbrace{\frac{\partial c}{\partial \mathbf{W}}}_{D \times H} = \underbrace{\mathbf{X}^\top}_{D \times N} \underbrace{\frac{\partial c}{\partial \mathbf{Z}}}_{N \times H}, \quad (24)$$

qui pourra être codée en Python par un simple appel à une fonction réalisant un produit matriciel (@ en Numpy), et

$$\underbrace{\frac{\partial c}{\partial \mathbf{b}}}_{1 \times H} = \underbrace{\mathbf{1}_N^\top}_{1 \times N} \underbrace{\frac{\partial c}{\partial \mathbf{Z}}}_{N \times H}, \quad (25)$$

qui pourra être codée en Python par un simple appel à une fonction sommant les lignes de  $\partial c / \partial \mathbf{Z}$  (.sum() en Numpy).

Les différentes expressions obtenues au cours de ce TD seront utilisées au prochain TP pour implémenter un MLP en Python (à l'aide de la bibliothèque Numpy).