TIPE - Les réseaux de neurone et leur application en reconnaissance d'images

Louis Caubet, Bastian Dupoirieux, Inès Vignal

2019

1 Introduction

L'intelligence artificielle, et plus précisément les réseaux de neurones, permettent de résoudre des problèmes complexes à un très grand nombre de paramètres, hors de portée d'un algorithme. Un espace aussi vaste et imprévisible que l'océan est un domaine qui correspond à ces caractéristiques. La start-up *Plastic Tide*, par exemple, utilise des drones munis de caméras équipés de réseaux de neurones capables de reconnaitre des images afin de repérer le plastique qui s'accumule à la surface des océans pour faciliter sa récupération.

Pour comprendre comment reconnaitre des images avec un réseau de neurone, nous nous intéresserons au problème (plus simple) de la reconnaissance de chiffres manuscrits. Cet exemple présente plusieurs avantages dans le cadre d'un TIPE :

- Une résolution satisfaisante nécessite des réseaux de neurones.
- Suffisamment simple pour ne pas nécessiter de puissance de calcul trop importante.
- Abondance en ligne de données d'images d'entraînement pour le réseau, prétraitées pour en simplifier la manipulation. (MNIST)

Table des matières

2 Construction du neurone et du réseau de neurones

2.1 Le neurone

a) Fonction d'activation

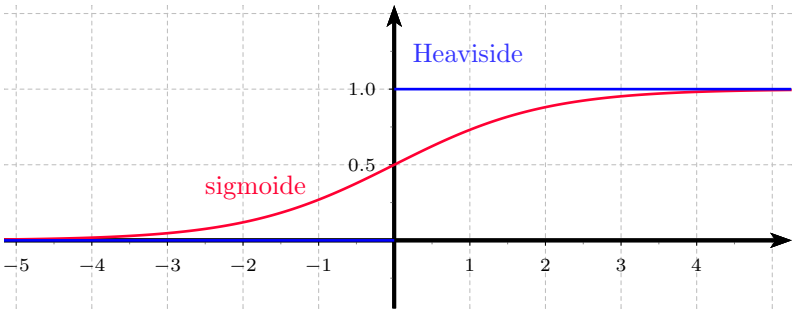
La fonction d'activation sert à transformer la sortie du neurone, dans $\mathbb{R}$ à priori, vers une valeur de $[0,1]$ qui pourra être utilisée comme entrée du neurone suivant. Une fonction d'activation naturelle est la fonction de Heaviside :

$$Heaviside : \begin{cases} \mathbb{R} \longrightarrow [0,1] \\ x \longmapsto \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases} \end{cases}$$

Celle-ci est une bonne modélisation d'un neurone biologique, mais ne permet pas l'utilisation de la méthode que nous allons voir après pour entrainer le réseau. Pour cela, il faut que la fonction d'activation soit dérivable. On choisit donc comme fonction d'activation la fonction sigmoïde :

$$\sigma : \begin{cases} \mathbb{R} \longrightarrow [0,1] \\ x \longmapsto \frac{1}{1+e^{-x}} \end{cases}$$

Cette fonction a l'avantage d'avoir un comportement similaire à Heaviside, et d'être rapide à calculer.



Cette fonction a également comme caractéristiques d'être non linéaire, monotone et d'avoir un ensemble d'arrivée infini. Même si ces caractéristiques ne sont pas indispensables dans la suite, elles favorisent l'apprentissage.

b) Modèle du neurone

Le neurone est l'élément de base d'un réseau de neurone. Comme son analogue biologique, il comporte n entrées, chacune dans $[0,1]$, et une sortie dans $[0,1]$. Il est caractérisé par n poids $(w_1, ..., w_n) \in \mathbb{R}^n$ ainsi qu'un biais $b \in \mathbb{R}$ et est modélisé par la fonction suivante :

$$\text{Neurone} : \begin{cases} [0,1]^n \longrightarrow [0,1] \\ (x_1, ..., x_n) \longmapsto \sigma(b + \sum_{i=1}^{n} w_i \times x_i) \end{cases}$$

Autrement dit, on calcule la combinaison linéaire des entrées pondérée par les poids, puis on applique la fonction d'activation au résultat.
Pour effectuer ce calcul plus rapidement, on calculera vectoriellement en posant $X = (x_1, ..., x_n)^T$ et $W = (w_1, ..., w_n)^T$.
Le calcul devient alors :

$$\text{Neurone} : \begin{cases} [0,1]^n \longrightarrow [0,1] \\ X = (x_1, ..., x_n) \longmapsto \sigma(W \cdot X + b) \end{cases}$$

2.2 Neurones en réseau simple : Multi-Layer Perceptron

On appelle perceptron multi-couche (*Multi-Layer Perceptron* en anglais), un réseau de neurones construit de la manière suivante :

— La couche d'entrée prend autant d'entrées que nécessaire pour représenter les données (par exemple, une entrée par pixel pour les images en noir et blanc que nous utiliserons). Chacune de ces entrées est normalisée dans $[0, 1]$.

— Les couches cachées sont constituées de n neurones, prenant chacun en entrée les sorties de tous les neurones de la couche précédente.

— La couche de sortie est constituée d'autant de neurones que de catégories de classification souhaitées.

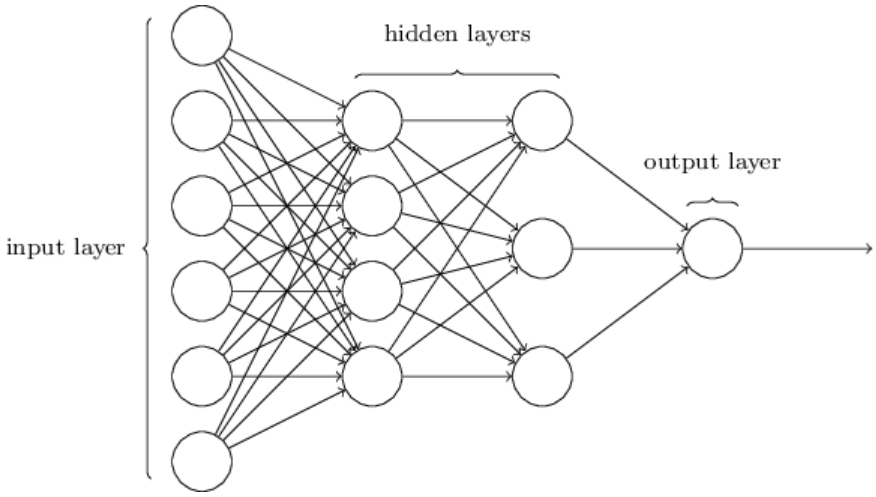


Figure : [2, Chapitre 1]

Le calcul de la sortie de ce réseau à partir d'une entrée se fait par la fonction feedforward $f_{\mathbf{w}}$, qui est la composition de la fonction Neurone selon la structure du réseau.

L'objectif va être d'entrainer ce réseau (c'est-à-dire trouver les valeurs des poids et biais de chaque neurone) pour atteindre l'objectif souhaité, par exemple reconnaitre des chiffres manuscrits. Pour cela, nous allons avoir besoin d'un certain nombre d'outils.

3 Outils mathématiques

On se placera ici dans l'espace euclidien $E = \mathbb{R}^n$.
On note $\langle a \mid b \rangle$ le produit scalaire usuel et $||\cdot||$ la norme associée.

3.1 Différentielle et dérivées partielles

Définition : Différentielle d'une fonction de E
On considère $f : E \longrightarrow \mathbb{R}$ et $a \in E$. f est différentiable en a s'il existe une application linéaire $L : E \longrightarrow F$ vérifiant :

$$\lim_{\substack{h \to 0_E \\ h \neq 0_E}} \frac{|f(a+h) - f(a) - L(h)|}{||h||} = 0$$

L'application L est alors unique et appelée différentielle de f en a et notée $\mathrm{d}f_a$.

Définition : Dérivée partielle

On considère $f : \begin{cases} E \longrightarrow \mathbb{R} \\ X = (x_1, ..., x_n) \longmapsto f(x_1, ..., x_n) \end{cases}$ et $a = (a_1, ..., a_n) \in E$.

La dérivée partielle de f en a par rapport à x_j, pour $j \in [\![1, n]\!]$, est, quand elle existe, notée $\dfrac{\partial f}{\partial x_j}(a)$ et définie par :

$$\frac{\partial f}{\partial x_j}(a) = \lim_{h \to 0} \frac{f(a_1, ..., a_{j-1}, a_j + h, a_{j+1}, ..., a_n) - f(a)}{h}$$

Définition / Proposition : Gradient

On considère $f : E \longrightarrow \mathbb{R}$ et $a \in E$. On suppose que f est différentiable en a. Le gradient de f en a est alors l'unique vecteur noté $\nabla_a f$ ou $\nabla f(a)$ et vérifiant

$$\forall u \in E, \mathrm{d}f_a(u) = \langle \nabla_a f \mid u \rangle$$

On montre alors que l'expression dans la base canonique de ce vecteur est

$$\nabla_a f = \left(\frac{\partial f}{\partial x_1}(a), ..., \frac{\partial f}{\partial x_n}(a) \right)^T$$

Proposition : CS de différentiabilité

On considère $f : E \longrightarrow \mathbb{R}$ et $a \in E$.
Si les n dérivées partielles de f sont définies sur un voisinage de a et continues en a, alors f est différentiable en a.

Proposition : Développement limité à l'ordre 1 dans E

On considère $f : E \longrightarrow \mathbb{R}$ différentiable en $a \in E$. f admet alors en a un développement limité à l'ordre 1 donné par :

$$f(a + h) = f(a) + \mathrm{d}f_a(h) + o(||h||)$$

Ceci s'écrit aussi :

$$f(a + h) = f(a) + \langle \nabla_a f \mid h \rangle + o(||h||)$$

4 L'entrainement du réseau de neurones

Grâce aux outils mathématiques que nous venons d'introduire, nous allons pouvoir démontrer un algorithme permettant d'entrainer ce réseau. Mais avant cela, il va falloir quantifier l'objectif que notre réseau doit atteindre

4.1 La fonction coût

La fonction coût va nous permettre d'évaluer la performance de notre réseau sur un ensemble de données d'entrainement, tout en vérifiant les conditions de différentiabilité et de convexité dont nous allons avoir besoin après.
Le minimum de cette fonction devra correspondre au fonctionnement optimal du réseau.
Nous allons utiliser ici la moyenne des erreurs au carré, appelée *Mean Squared Error* ou *MSE*. Elle est définie de la manière suivante :

$$\text{MSE} : \begin{cases} \mathbb{R}^n \longrightarrow \mathbb{R} \\ (a_1, ..., a_n) \longmapsto \dfrac{1}{2n} \displaystyle\sum_{i=1}^{n} ||a_i - y_i||^2 \end{cases}$$

avec :

- n le nombre d'éléments dans l'ensemble d'entrainement
- a_i la sortie du réseau avec le ième élément d'entrainement pour entrée
- y_i la sortie attendue du réseau avec le ième élément d'entrainement pour entrée

Nous allons montrer que cette fonction est différentiable sur $\mathbb{R}^n$.

Démonstration :

Montrons que la fonction *MSE* est différentiable par la caractérisation de la différentiabilité.
Montrons que *MSE* est differentiable en tout point de $\mathbb{R}^n$

Soit $i \in [\![1, n]\!]$. et A=$(a_1, ..., a_n) \in \mathbb{R}^n$.
Montrons que $\dfrac{\partial MSE}{\partial x_i}$ est bien définie au voisinage de A et continue en A.

Par rapport à x_i, MSE est une somme de constantes et de $\dfrac{1}{2n}||a_i - y_i||^2$, donc d'une norme au carré, donc d'un produit scalaire. Comme $a_i - y_i$ est C^1, $\langle a_i - y_i \mid a_i - y_i \rangle$ aussi (dérivée d'un produit).

Conclusion : $\dfrac{\partial MSE}{\partial x_i}$ est bien définie sur $\mathbb{R}^n$ et est continue.
Donc *MSE* est différentiable sur $\mathbb{R}^n$.

On peut alors définir la fonction qui associe le coût directement au réseau (et non aux sorties) :

$$\mathrm{C} : \begin{cases} \mathbb{R}^n \longrightarrow \mathbb{R} \\ \mathbf{w} \longmapsto \dfrac{1}{2n} \sum\limits_{i=1}^{n} ||f_{\mathbf{w}}(x_i) - y_i||^2 \end{cases}$$

avec $\mathbf{w}$ les paramètres du réseau (poids et biais).

$f_{\mathbf{w}}$ est différentiable car la fonction Neurone l'est, et $f_{\mathbf{w}}$ est une simple composition de ces fonctions.
De plus, MSE est différentiable donc C est différentiable.

4.2 L'algorithme de descente du gradient

Lemme : Direction de plus forte descente
On considère $f : E \longrightarrow \mathbb{R}$ différentiable, et $a \in E$. La direction de plus forte descente de f en a est celle du vecteur $u \in \{h \in E \ / \ ||h|| = 1\}$ qui minimise $f(a + u)$. On va montrer que cette direction est celle opposée au gradient de f en a, ie. $u = \dfrac{-\nabla_a f}{||\nabla_a f||}$.

Démonstration : La formule de Taylor traduit le fait que, pour $h \in E$ avec $||h|| = 1$, minimiser $f(a + h)$ revient à minimiser $\mathrm{d}f_a(h)$.
On a vu que $\mathrm{d}f_a(h) = \langle \nabla_a f \mid h \rangle$. On cherche donc le vecteur h de norme 1 qui minimise $\langle \nabla_a f \mid h \rangle$.

Par l'inégalité de Cauchy-Schwarz, on a :

$$|\langle \nabla_a f \mid h \rangle| \leq ||\nabla_a f|| \cdot ||h||$$

Avec $||h|| = 1$, donc en particulier :

$$\langle \nabla_a f \mid h \rangle \geq -||\nabla_a f||$$

L'égalité est atteinte pour $h = \dfrac{-\nabla_a f}{||\nabla_a f||}$, CQFD.

Algorithme de descente du gradient

On se place sur l'espace euclidien E et on considère une fonction $f : E \longrightarrow \mathbb{R}$ différentiable. L'algorithme de descente du gradient va nous permettre de trouver un minimum local de f.

Le principe est de se déplacer dans la direction de plus forte descente à chaque itération, jusqu'à ce qu'un test d'arrêt soit vérifié.

On considère $x_0 \in E$ un point de départ, un seuil de tolérance $\varepsilon > 0$, et un pas d'apprentissage $\eta > 0$.
Tant que $||\nabla f(x_k)|| > \varepsilon$, on construit $x_{k+1} = x_k - \eta \cdot \nabla f(x_k)$.

Cet algorithme ne peut se démontrer qu'avec des hypothèses de régularité plus fortes sur la fonction f (f doit être convexe et à gradients lipschitziens) [8]. Nous nous contenterons donc d'une démonstration graphique et d'une vérification expérimentale.

Le choix de η est essentiel à la convergence de l'algorithme. Si η est trop grand, l'algorithme diverge, et si η est trop petit, l'algorithme converge très lentement, comme montré sur la figure suivante.

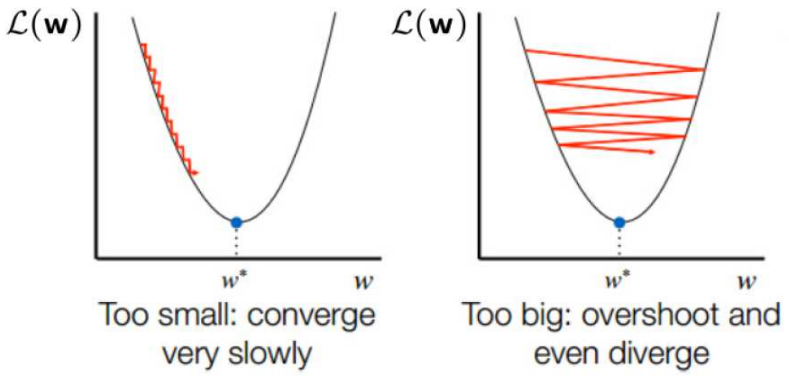


Figure : [9, Séquence 1, Chapitre 4]

Comme on l'a vu en lemme, la direction de plus forte descente est celle opposée au gradient. Ainsi, par cet algorithme, pour η bien choisi, on va se déplacer de la manière suivante, jusqu'à atteindre un minimum local.

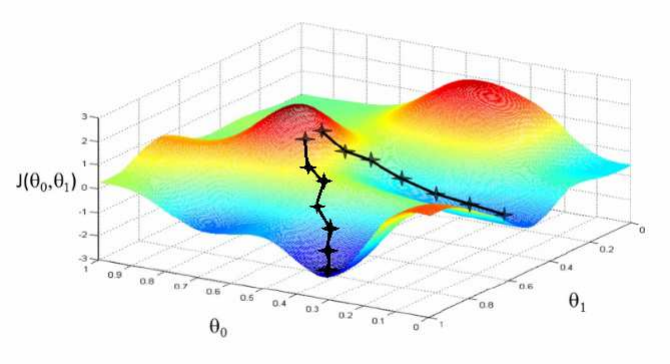


Figure : [9, Séquence 1, Chapitre 4]

Il reste quand même un défaut majeur à cet algorithme quand on veut travailler avec des grands ensembles d'entrainement : à chaque itération, il faut calculer les gradients par rapport à chaque entrée, ce qui peut être très long.
En effet, si on note $C_i = \frac{1}{2}||f_{\mathbf{w}}(x_i) - y_i||^2$ pour tout $i \in [\![1; n]\!]$, on a $\nabla C = \frac{1}{n}\sum_{i=1}^{n} \nabla C_i$.
On cherche donc un moyen d'optimiser ceci, et on introduit pour cela une variante de l'algorithme de descente de gradient.

4.3 Descente du gradient stochastique (SGD)

La méthode de descente de gradient stochastique (SGD) permet d'accélérer avec une bonne estimation la phase d'apprentissage de l'algorithme : au lieu de calculer le gradient par rapport à chaque entrée, on le calcule par rapport à un *mini-batch* ou *mini-lot* de m entrées $(x_1, \dots, x_m)$ choisies aléatoirement (d'où l'aspect "stochastique").

Pour un m suffisamment grand (on le constate empiriquement), on a :

$$\frac{1}{m}\sum_{j=1}^{m}\nabla C_{x_j} \approx \frac{1}{n}\sum_{i=1}^{n}\nabla C_{x_i} = \nabla C$$

Ainsi, dans l'expression :

$$x_{k+1} = x_k - \eta \cdot \nabla C$$

on peut remplacer ∇C par $\frac{1}{m}\sum_{j=1}^{m}\nabla C_{x_j}$, bien plus rapide à calculer. Par exemple, nous utiliserons un *training set* de 60 000 images. En choisissant à chaque fois un *mini-batch* de 10 images, notre algorithme sera 6 000 fois plus rapide sans trop perdre en précision.

On entraine successivement l'algorithme avec des mini-batchs aléatoires jusqu'à épuisement des données d'entrainement : la première *epoch* est alors terminée !

4.4 Rétropropagation du gradient

Il ne nous manque plus qu'une chose pour pouvoir entrainer notre réseau : comment calculer le gradient de la fonction coût, par rapport aux poids et biais du réseau ?

Pour cela, nous allons utiliser l'algorithme de rétropropagation du gradient. L'idée de cet algorithme est de remonter à l'origine de l'erreur depuis la fin du réseau, en déterminant pour chaque neurone, en quoi chaque valeur est responsable de l'erreur. Mathématiquement, cela revient exactement à calculer les dérivées partielles.

On s'intéresse ici au réseau dans le cas d'une entrée $x = (x_1, ..., x_p)$. On note $y = (y_1, ..., y_q)$ la sortie attendue et $a^L = (a_1, ..., a_q)$ la sortie obtenue. On note également $C = \frac{1}{2}||a - y||^2 = \frac{1}{2}\sum_{j=1}^{q}(a_j - y_j)^2$ la fonction coût partielle selon x.

Définition : Produit de Hadamard
On note $A \circ B$ le produit terme à terme des matrices A et B, avec $(A, B) \in M_{n,p}(\mathbb{R})^2$. Le produit de Hadamard est associatif, commutatif et distributif. L'élément neutre pour $\circ$ est bien entendu la matrice ne contenant que des coefficients 1.

Notations :
Nous allons avoir besoin de noter sans ambiguïté les poids et biais du réseau. Dans toute la suite, nous utiliserons donc les notations suivantes :

- On numérote les couches du réseau de 1 à L.
- Dans une couche l, on numérote les neurones de 1 à p_l
- Pour un certain neurone k de la l-ième couche, on note w^l_{kj} le poids correspondant à la j-ième entrée de ce neurone, et b^l_k le biais de ce neurone.
- On note a^l_k la sortie du neurone k de la couche l.
- On note z^l_k l'entrée pondérée du neurone k de la couche l, ie. $z^l_k = \sum_j (w^l_{kj} \times a^{l-1}_j) + b^l_k$
- Pour une valeur x définie sur chaque neurone, on note x^l la matrice colonne des x^l_k les valeurs de x pour les neurones de la couche l.

Définition : Erreur
On définit l'erreur en un neurone (l, k) par

$$\delta_k^l = \frac{\partial C}{\partial z_k^l}$$

L'algorithme de rétropropagation nous permettra de calculer cette erreur pour chaque neurone. On pourra ensuite en déduire les valeurs qui nous intéressent, ie. les dérivées partielles par rapport aux poids et biais.

Équations de la rétropropagation du gradient
Avec les notations précédentes, on a les relations suivantes :

L'équation nous donnant l'erreur en sortie :

$$\delta_k^L = \frac{\partial C}{\partial a_k^L} \cdot \sigma'(z_k^L) \tag{1a}$$

Ceci peut se réécrire de manière vectorielle de la façon suivante :

$$\delta^L = \nabla C \circ \sigma'(z^L) \tag{1b}$$

Ou encore, dans le cas de la fonction coût MSE, avec $\nabla C = (a^L - y)$, on a

$$\delta^L = (a^L - y) \circ \sigma'(z^L) \tag{1c}$$

L'équation exprimant l'erreur δ^l en fonction de celle à la couche suivante, δ^{l+1}

On note w^{l+1} la matrice suivante :

$$w^{l+1} = \begin{bmatrix} w_{11}^{l+1} & . & . & . & w_{p1}^{l+1} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ w_{1p}^{l+1} & . & . & . & w_{pp}^{l+1} \end{bmatrix}$$

On a :

$$\delta^l = (w^{l+1} \times \delta^{l+1}) \circ \sigma'(z^l) \tag{2}$$

L'équation exprimant les dérivées partielles par rapport aux biais

$$\frac{\partial C}{\partial b_k^l} = \delta_k^l \tag{3}$$

L'équation exprimant les dérivées partielles par rapport aux poids

$$\frac{\partial C}{\partial w_{kj}^l} = a_j^{l-1} \times \delta_k^l \tag{4}$$

Démonstration :

On admettra pour ces démonstrations la règle de la chaine pour les fonctions de plusieurs variables : pour $f : \begin{cases} \mathbb{R}^n \longrightarrow \mathbb{R} \\ (u_1, ..., u_n) \longmapsto f(u_1, ..., u_n) \end{cases}$ *et* $\forall i \in [\![1, n]\!], u_i = g_i(x_1, ..., x_p)$, *avec* $g_i : \mathbb{R}^p \longrightarrow \mathbb{R}$, *on a la relation suivante :* $\frac{\partial f}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial u_j} \cdot \frac{\partial u_j}{\partial x_i}$

Montrons (1) :

$$\begin{aligned}
\delta_k^L &= \frac{\partial C}{\partial z_k^L} && \text{(Définition)} \\
&= \sum_{j=1}^{p_L} \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_k^L} && \text{(Règle de la chaine)} \\
&= \frac{\partial C}{\partial a_k^L} \cdot \frac{\partial a_k^L}{\partial z_k^L} && \text{(Pour } j \neq k, \frac{\partial a_j^L}{\partial z_k^L} = 0 \text{ car la sortie du neurone j ne dépend que de l'entrée du neurone j)} \\
&= \frac{\partial C}{\partial a_k^L} \cdot \sigma'(z_k^L) && (a_k^L = \sigma(z_k^L))
\end{aligned}$$

Montrons (2) :

$$\begin{aligned}
\delta_k^l &= \frac{\partial C}{\partial z_k^l} && \text{(Définition)} \\
&= \sum_{j=1}^{p_{l+1}} \frac{\partial C}{\partial z_j^{l+1}} \cdot \frac{\partial z_j^{l+1}}{\partial z_k^l} && \text{(Règle de la chaine)} \\
&= \sum_{j=1}^{p_{l+1}} \delta_j^{l+1} \cdot \frac{\partial z_j^{l+1}}{\partial z_k^l} && \text{(Définition de } \delta_j^{l+1})
\end{aligned}$$

Soit $j \in [\![1, p]\!]$. On a :

$$z_j^{l+1} = \sum_{i=1}^{p_l} w_{ji}^{l+1} a_i^l + b_j^{l+1} = \sum_{i=1}^{p_l} w_{ji}^{l+1} \sigma(z_i^l) + b_j^{l+1}$$

On dérive par rapport à z_k^l :

$$\frac{\partial z_j^{l+1}}{\partial z_k^l} = w_{jk}^{l+1} \sigma'(z_k^l)$$

On en déduit :

$$\delta_k^l = \sum_{j=1}^{p_{l+1}} w_{jk}^{l+1} \cdot \sigma'(z_k^l) \cdot \delta_j^{l+1}$$

Ce qui est exactement le coefficient k de (2).

Montrons (3) :

$$\begin{aligned}
\frac{\partial C}{\partial b_k^l} &= \sum_{j=1}^{p_l} \frac{\partial C}{\partial z_j^l} \cdot \frac{\partial z_j^l}{\partial b_k^l} && \text{(Règle de la chaine)} \\
&= \frac{\partial C}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial b_k^l} && \text{(Pour } k \neq j, \frac{\partial z_j^l}{\partial b_k^l} = 0)
\end{aligned}$$

Or, on a :

$$\frac{\partial z_k^l}{\partial b_k^l} = \frac{\partial \left(\sum_{i=1}^{p_{l-1}} w_{ki}^l a_i^{l-1} + b_k^l \right)}{\partial b_k^l} = 1$$

D'où le résultat :

$$\frac{\partial C}{\partial b_k^l} = \delta_k^l$$

Montrons (4) :

$$\begin{aligned}
\frac{\partial C}{\partial w_{kj}^l} &= \sum_{i=1}^{p_l} \frac{\partial C}{\partial z_i^l} \cdot \frac{\partial z_i^l}{\partial w_{kj}^l} && \text{(Règle de la chaine)} \\
&= \frac{\partial C}{\partial z_k^l} \cdot \frac{\partial z_k^l}{\partial w_{kj}^l} && \text{(Pour } i \neq k, \frac{\partial z_i^l}{\partial w_{kj}^l} = 0\text{)} \\
&= \delta_k^l \cdot \frac{\partial \left(\sum_{i=1}^{p_{l-1}} w_{ki}^l \cdot a_i^{l-1} + b_k^l \right)}{\partial w_{kj}^l} \\
&= \delta_k^j \cdot a_j^{l-1}
\end{aligned}$$

L'algorithme de rétropropagation du gradient

Avec ces équations, nous pouvons écrire un algorithme calculant le gradient de la fonction coût :

Entrée

Prendre un vecteur x en entrée. Déterminer l'activation de la première couche a^1 correspondante.

Phase de propagation vers l'avant

Pour toute couche $l \in [\![2, L]\!]$, calculer z^l et $a^l = \sigma(z^l)$.

Erreur en sortie

Calculer le vecteur $\delta^L = \nabla C \circ \sigma'(z^L)$ (Formule (1)).

Phase de rétropropagation de l'erreur

Pour toute couche l de $L-1$ à 2, calculer l'erreur $\delta^l = (w^{l+1} \times \delta^{l+1}) \circ \sigma'(z^l)$ (Formule (2))

Sortie

Le gradient de la fonction coût est donné par :

$\forall l \in [\![2, L]\!], \forall k \in [\![1, p_l]\!]$,

$\frac{\partial C}{\partial b_k^l} = \delta_k^l$ (Formule (3))

$\forall j \in [\![1, p_{l-1}]\!], \frac{\partial C}{\partial w_{kj}^l} = a_j^{l-1} \cdot \delta_k^l$ (Formule (4))

5 Implémentation du réseau de neurones en Python

L'implémentation est essentiellement la traduction des formules introduites précédemment, avec quelques subtilités pour obtenir un programme suffisamment rapide, notamment l'utilisation de numpy/scipy.

5.1 La construction du réseau de neurones

On utilise un objet Reseau qui contiendra toutes les méthodes dont nous aurons besoin pour entrainer puis évaluer notre réseau. Le code de construction du réseau se situe bien entendu dans la méthode __init__, qui prend 4 arguments :

- nb_couches le nombre de couches cachées du résau
- neurones_par_couche le nombre de neurones par couche cachée
- nb_entrees le nombre de neurones sur la couche d'entrée
- nb_sorties le nombre de neurones sur la couche de sortie

Chaque couche est représentée par une matrice des poids de taille (nombre de sorties de la couche) $\times$ (nombre d'entrées de la couche) et une matrice de biais de taille (nombre de sorties de la couche) $\times 1$. Ces matrices sont initialisées avec des valeurs aléatoires grâce à la fonction random.randn de numpy et stockées dans des listes poids et biais.

```
1  def __init__(self, nb_couches, neurones_par_couche, nb_entrees, nb_sorties):
2    # poids est une liste de matrices contenant les poids pour chaque couche
3    self.poids = list()
4    # idem pour biais
5    self.biais = list()
6
7    # Copie des parametres en champs.
8    self.nb_couches, self.neurones_par_couche, self.nb_entrees, self.nb_sorties = nb_couches,
         neurones_par_couche, nb_entrees, nb_sorties
9
10
11   # Initialisation avec des valeurs aleatoires
12
13   # Premiere couche
14   self.poids.append(np.random.randn(neurones_par_couche, nb_entrees))
15   self.biais.append(np.random.randn(neurones_par_couche, 1))
16
17   # couches cachees
18   for couche in range(1, nb_couches):
19     self.poids.append(np.random.randn(neurones_par_couche, neurones_par_couche))
20     self.biais.append(np.random.randn(neurones_par_couche, 1))
21
22   # couche de sortie
23   self.poids.append(np.random.randn(nb_sorties, neurones_par_couche))
24   self.biais.append(np.random.randn(nb_sorties, 1))
```

5.2 Propagation vers l'avant

Pour la couche d'entrée, les images sont en noir et blanc donc on a une valeur par pixel comprise entre 0 et 255. On va utiliser une fonction d'activation différente pour cette première couche, pour ramener cette valeur dans $[0, 1]$.

```
1  def normalise_entree(self, entree):
2    """Fonction d'activation particuliere pour transformer l'entree dans [0,1]. """
3    return entree / 255
```

Pour l'implémentation de la fonction sigmoide, on pourrait penser l'implémenter naivement et utiliser np.vectorize Les résultats sont en réalité très mauvais avec cette méthode. On va utiliser la fonction sigmoide vectorielle déjà implémentée dans le module scipy sous le nom expit, et on la renomme en sigmoid.

```
1  from scipy.special import expit as sigmoid
```

Pour exécuter feedforward rapidement, on remarque qu'on peut faire les calculs matriciellement couche par couche de la façon suivante :

$$\begin{bmatrix} w^l_{11} & . & . & w^l_{1p} \\ . & & & . \\ . & & & . \\ w^l_{p1} & . & . & w^l_{pp} \end{bmatrix} \times \begin{bmatrix} a^{l-1}_1 \\ . \\ . \\ a^{l-1}_p \end{bmatrix} + \begin{bmatrix} b^l_1 \\ . \\ . \\ b^l_p \end{bmatrix} = \begin{bmatrix} z^l_1 \\ . \\ . \\ z^l_p \end{bmatrix}$$

L'implémentation, en conservant les résultats de chaque étape, dont nous aurons besoin pour la rétropropagation est la suivante :

```
def feedforward(self, entree):
  """Determine la sortie du reseau pour l'entree, en memorisant les entrees ponderees et
      sorties intermediaires."""

  # Stocke les entrees ponderees
  entrees_ponderees = []
  # Stocke les sorties
  sorties = [self.normalise_entree(entree)]

  # Propagation
  for couche in range(self.nb_couches+1):
    entrees_ponderees.append(np.dot(self.poids[couche], sorties[couche])+self.biais[couche])
    sorties.append(sigmoid(entrees_ponderees[couche]))

  return (entrees_ponderees, sorties)
```

Pour l'utilisation du réseau hors de la phase d'entrainement, on implémente également une méthode qui ne renvoie que le résultat :

```
def execute(self, entree):
  _, sorties = self.feedforward(entree)
  return sorties[self.nb_couches + 1]
```

5.3 Rétropropagation du gradient

On commence par implémenter la fonction dérivée de sigmoide :

```
def sigmoid_prime(x):
  return sigmoid(x)*(1-sigmoid(x))
```

Ensuite, c'est simplement la traduction de l'algorithme présenté plus haut, en profitant de la capacité de Python à utiliser des indices de liste négatifs.

```
def retropropagation(self, entree, sortie_attendue):
  """Determine le gradient de la fonction cout par l'algorithme de retropropagation"""

  # Calcul du gradient

  # nabla_w est une liste de matrices (meme forme que poids mais avec des derivees partielles)
  nabla_w = [np.zeros(np.shape(w)) for w in self.poids]

  # nabla_b est de la forme de biais mais avec des derivees partielles
  nabla_b = [np.zeros(np.shape(b)) for b in self.biais]


  # Phase 1 : Feedforward
  (entrees_ponderees, sorties) = self.feedforward(entree)

  # Erreur en sortie
  erreur = (sorties[-1] - sortie_attendue) * sigmoid_prime(entrees_ponderees[-1])

  # Gradient en sortie
  nabla_b[-1] = erreur
  nabla_w[-1] = np.dot(erreur, sorties[-2].transpose())

```

```
23
24    # Phase 2 : Retropropagation
25    for couche in range(2, self.nb_couches + 2):
26      # Retropropagation de l'erreur
27      erreur = np.dot(self.poids[-couche+1].transpose(), erreur) *
            sigmoid_prime(entrees_ponderees[-couche])
28      # Calcul du gradient
29      nabla_w[-couche] = np.dot(erreur, sorties[-couche-1].transpose())
30      nabla_b[-couche] = erreur
31
32    return (nabla_w, nabla_b)
```

5.4 Descente du gradient

L'implémentation de l'algorithme de descente de gradient est très simple grâce à la forme matricielle que nous utilisons : c'est simplement une soustraction de matrices sur chaque couche. Avant ça, il faut calculer le gradient sur le batch entier. La rétropropagation nous permet de le calculer pour chaque élément du batch, il suffit donc de faire la moyenne. On obtient l'algorithme suivant :

```
1  def descente_gradient(self, batch, eta):
2    """Applique la descente de gradient sur le reseau avec batch comme set d'entrainement et
         eta comme pas d'apprentissage"""
3
4    # On calcule la moyenne des gradients partiels sur batch
5    grad_w = [np.zeros(w.shape) for w in self.poids]
6    grad_b = [np.zeros(b.shape) for b in self.biais]
7
8    for i in range(len(batch)):
9      (entree, sortie_attendue) = batch[i]
10     nabla_w, nabla_b = self.retropropagation(entree, sortie_attendue)
11
12     grad_w = [grad_w[j] + nabla_w[j] for j in range(len(grad_w))]
13     grad_b = [grad_b[j] + nabla_b[j] for j in range(len(grad_b))]
14
15
16   # On applique la descente de gradient
17   self.poids = [self.poids[i] - (eta/len(batch))*grad_w[i] for i in range(len(self.poids))]
18   self.biais = [self.biais[i] - (eta/len(batch))*grad_b[i] for i in range(len(self.biais))]
```

5.5 Entrainement du réseau avec la SGD

Avec toutes les méthodes précédentes, on peut implémenter la descente de gradient stochastique. Pour mélanger l'ensemble d'entrainement, on utilise la fonction random.shuffle. Le code est ensuite une simple boucle pour exécuter la descente de gradient sur chacun des batchs. Après chaque epoch, on évalue la performance du réseau avec la méthode présentée plus bas. les autres appels à print servent à afficher l'avancement de l'algorithme.

```
1  def SGD(self, entrainement, test, epochs, taille_batch, eta):
2    """Execute une descente de gradient stochastique avec les arguments suivants:
3      - entrainement: l'ensemble des donnees d'entrainement.
4      - test: ensemble des donnees de validation """
5
6    # On evalue le reseau genere aleatoirement
7    print("Score du reseau aleatoire :", self.evalue(test), "/", len(test))
8
9    for i in range(epochs):
10     print("Epoch:", i)
11     # On melange le set d'entrainement
12     random.shuffle(entrainement)
13     # On le decoupe en batchs d'entrainement
14     batchs = [entrainement[k:k+taille_batch] for k in range(0, len(entrainement),
           taille_batch)]
15     # On execute la descente de gradient sur chacun des batchs
16     for j in range(len(batchs)):
17       self.descente_gradient(batchs[j], eta)
18       if j%100 == 0:
19         print("    Batch:", j, "/", len(batchs), end="\r")
20
21     print("    Batch:", len(batchs), "/", len(batchs), end="\n")
22     print("Score:", self.evalue(test), "/", len(test))
```

5.6 Évaluation du réseau

Pour évaluer le réseau sur un ensemble de test, il suffit de l'exécuter sur chacune des entrées et de comparer avec la sortie attendue, puis de compter le nombre d'entrées pour laquelle la sortie était correcte. L'implémentation est la suivante :

```
def evalue(self, set_test):
    """Evalue la performance du reseau sur le set de test."""

    test_results = [(np.argmax(self.execute(x)), y) for (x,y) in set_test]
    return sum(int(x == y) for x,y in test_results)
```

5.7 Import et traitement de la base de données MNIST

La base de données MNIST est un ensemble d'images de chiffres manuscrits spécifiquement conçue pour entrainer des réseaux de neurones. Le module Python mnist permet de charger celle-ci facilement, mais les données obtenues ne sont pas totalement prêtes à entrainer notre réseau. Nous allons donc devoir les modifier.
Plus précisément, il faut séparer l'ensemble d'entrainement en ensemble d'entrainement et ensemble de validation, et dans l'ensemble d'entrainement, remplacer les sorties qui sont des entiers en matrices 10×1 contenant un 1 sur la ligne d'indice le chiffre souhaité. Il faut également transformer les listes d'entrées et de sorties attendues en une liste de couples (entrée, sortie attendue). C'est ce que réalise le code suivant :

```
def load_mnist():

    # On charge les fichiers de MNIST avec la bibliotheque mnist.
    mndata = MNIST('./data')
    mndata.gz = True
    img_entr, labels_entr = mndata.load_training()
    img_test, labels_test = mndata.load_testing()

    # On va transformer les donnees brutes en donnees adaptees a notre algorithme d'entrainement

    # On decoupe l'ensemble 'training' en set d'entrainement et set de validation
    split = int(len(img_entr) * 0.9)
    entrees_entrainement = [np.reshape(img_entr[i], (784,1)) for i in range(split)]
    resultats_entrainement = [matrice_resultat(labels_entr[i]) for i in range(split)]

    entrainement = list(zip(entrees_entrainement, resultats_entrainement))

    entrees_validation = [np.reshape(img_entr[i], (784,1)) for i in range(split, len(img_entr))]
    resultats_validation = [labels_entr[i] for i in range(split, len(img_entr))]

    validation = list(zip(entrees_validation, resultats_validation))

    entrees_test = [np.reshape(img_test[i], (784,1)) for i in range(len(img_test))]

    evaluation = list(zip(entrees_test, labels_test))

    return (entrainement, validation, evaluation)


def matrice_resultat(j):
    """Renvoie un vecteur 10x1 avec un coefficient 1 a la je position et des zeros ailleurs.
        Ceci permet de convertir un chiffre en sortie attendue du reseau."""
    e = np.zeros((10, 1))
    e[j] = 1.0
    return e
```

5.8 Exécution du réseau et résultats

Le code pour construire le réseau et exécuter l'apprentissage est simplement :

```
# On charge mnist
set_entrainement, set_validation, set_test = mnist_loader.load_mnist()


```

```
# on construit le reseau
reseau = Reseau(2, 15, 784, 10)

# on entraine le reseau sur 30 epochs, avec des mini-batchs de 10 et eta=3
reseau.SGD(set_entrainement, set_validation, 30, 10, 3)

# on evalue le reseau sur l'ensemble d'evaluation
print("Evaluation :", reseau.evalue(set_test), "/", len(set_test))
```

Le nombre d'epochs et la valeur de η ont été déterminées expérimentalement comme permettant d'obtenir des résultats concluants. Malgré toutes les optimisations faites, l'entrainement nécessite toujours plusieurs minutes sur une machine moderne.
Le résultat obtenu dépend de l'initialisation aléatoire. Nous avons obtenu les résultats suivants :

```
Score du reseau aleatoire : 590 / 6000
Epoch: 0
    Batch: 5400 / 5400
Score: 5498 / 6000
Epoch: 1
    Batch: 5400 / 5400
Score: 5564 / 6000
Epoch: 2
    Batch: 5400 / 5400
Score: 5625 / 6000
Epoch: 3
    Batch: 5400 / 5400
Score: 5593 / 6000

...

Epoch: 26
    Batch: 5400 / 5400
Score: 5695 / 6000
Epoch: 27
    Batch: 5400 / 5400
Score: 5709 / 6000
Epoch: 28
    Batch: 5400 / 5400
Score: 5644 / 6000
Epoch: 29
    Batch: 5400 / 5400
Score: 5696 / 6000
Evaluation : 9393 / 10000

Temps total :  705s
```

Avec ce réseau relativement simple - deux couches cachées de 15 neurones, on obtient un résultat de presque 94% de réussite sur l'ensemble d'évaluation. Même si le réseau est simple, l'entrainement est déjà long - plus de 10 minutes !
Avec 2 couches de 30 neurones, on parvient à dépasser 95% de réussite au prix d'un entrainement encore plus long, mais il est difficile de faire beaucoup mieux avec le modèle que nous avons utilisé ici.

```
Score du reseau aleatoire : 775 / 6000
Epoch: 0
    Batch: 5400 / 5400
Score: 5500 / 6000
Epoch: 1
    Batch: 5400 / 5400
Score: 5611 / 6000

...

Epoch: 28
    Batch: 5400 / 5400
Score: 5755 / 6000
Epoch: 29
    Batch: 5400 / 5400
Score: 5762 / 6000
Evaluation : 9532 / 10000

Temps total:  790s
```

6 Conclusion

Dans ce TIPE, nous avons donc introduit un certain nombre d'outils nécessaires à l'apprentissage des réseaux de neurones, notamment l'algorithme de descente du gradient et la rétropropagation. Nous avons ensuite implémentés ces algorithmes et, ensemble, ils forment un programme complet permettant l'entrainement et l'exécution d'un réseau de neurone, à partir d'un ensemble d'entrainement.
Après optimisation, principalement par l'utilisation de numpy, ce programme donne des résultats satisfaisants sur le problème de la reconnaissance de chiffres manuscrits.

On notera toutefois que 95% n'est en réalité pas un excellent score pour un réseau de neurones sur ce problème, et qu'on peut obtenir quelques pourcents de plus en augmentant la taille du réseau, en utilisant une autre fonction coût (l'entropie croisée) qui remédie à certains problèmes de la MSE au prix d'une plus faible intuitivité, mais qui est utilisée en pratique, et en utilisant des techniques de régularisation des poids. Nous n'avons pas eu le temps d'aborder toutes ces techniques ici, mais cela aurait été la suite logique pour améliorer nos résultats sur ce problème.

Enfin, nous avons passé sous silence un élément pourtant essentiel : le choix des hyper-paramètres, notamment le pas d'apprentissage η. Pour obtenir la valeur choisie, nous avons expérimenté avec des valeurs de différents ordres de grandeur, jusqu'à obtenir quelque chose de concluant, mais il existe des techniques qui permettent d'affiner ce choix et donc potentiellement d'obtenir de meilleurs résultats.

En ce qui concerne la recherche de plastique dans l'océan, le problème peut être résolu de la même manière, avec un réseau de neurones, mais nous avons ici ignoré la plus grande difficulté : obtenir les données d'entrainement. Ce n'était pas le cadre de ce TIPE, et par conséquent, nous nous sommes contentés d'utiliser une base de données déjà disponible, mais c'est la plus grande difficulté de toute entreprise qui cherche à résoudre un problème avec des réseaux de neurones.

7 Bibliographie

Références

[1] Briony Harris, The latest weapons in the fight against ocean plastic ? Drones and an algorithm, *weforum.org*, 2018.

[2] Michael Nielsen, Neural Networks and Deep Learning, *neuralnetworksanddeeplearning.com*, Consulté en mars 2019.

[3] Wikipédia, Différentielle, *fr.wikipedia.org/wiki/Différentielle*, Consulté le 06/03/2019.

[4] Wikipédia, Dérivée partielle, *fr.wikipedia.org/wiki/Dérivée_partielle*, Consulté le 06/03/2019.

[5] Wikipédia, Gradient, *fr.wikipedia.org/wiki/Gradient*, Consulté le 06/03/2019.

[6] Wikipédia, Théorème de Taylor, **Section :** Formules de Taylor pour les fonctions de plusieurs variables *fr.wikipedia.org/wiki/Théorème_de_Taylor*, Consulté le 06/03/2019.

[7] Wikipédia, Algorithme du gradient, *fr.wikipedia.org/wiki/Algorithme_du_gradient*, Consulté le 06/03/2019.

[8] Robert M. Gower, Convergence Theorems for Gradient Descent, 5 Octobre 2018, *perso.telecom-paristech.fr/rgower/pdf/M2_statistique_optimisation/grad_conv.pdf*.

[9] Nicolas Thome, CNAM, MOOC sur le Deep Learning, *fun-mooc.fr (Archive)*, Avril 2018.

[10] Wikipédia, Chain Rule (Anglais), *en.wikipedia.org/wiki/Chain_rule*, Consulté le 08/04/2019.

[11] Wikipédia, Base de données MNIST, *fr.wikipedia.org/wiki/Base_de_données_MNIST*, Consulté le 12/05/2019.

[12] Yann LeCun, Corinna Cortes, Christopher J.C. Burges, The MNIST Database, *yann.lecun.com/exdb/mnist*.

[13] Wikipédia, Réseau de neurones artificiel, *fr.wikipedia.org/wiki/Réseau_de_neurones_artificiels*, Consulté le 12/05/2019.