Expressivité des réseaux neuronaux

Théophile Baggio
Raphaël Dion
Yisakor Weldegebriel

Encadré par
Clément Marteau

Master 1 Mathématiques Appliquées, Statistiques

Université Lyon 1

2020/2021

Table des matières

1 Etude bibliographique sur les réseaux neuronaux

Un réseau de neurones peut être défini comme une série de noeuds (neurones) organisés sur plusieurs couches qui pour une entrée donnée produisent une sortie pondérée par les coefficients des neurones. Le réseau a la capacité de s'entrainer sur des données et de produire les sorties désirées par l'utilisateur. Il existe des applications multiples allant de l'approximation de fonctions à la reconnaissance d'images.

1.1 Premier article : Deep Learning : An Introduction for Applied Mathematicians[1]

L'objectif de cet article est d'introduire les principes de base du fonctionnement des réseaux de neurones et de les illustrer par des tâches de classification. La fonction d'activation utilisée dans ce contexte est une sigmoïde :

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

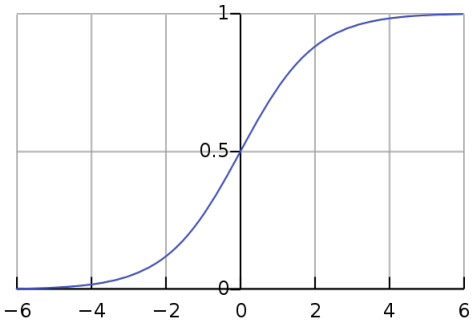

FIGURE 1 – Courbe d'une fonction sigmoïdale
[2]

Un réseau de neurones est composé de deux à plusieurs couches et ces couches sont elles-mêmes composées de neurones (nœuds). Essentiellement, il s'agit d'une fonction qui prend un vecteur en entrée pour ensuite calculer les coefficients de sortie grâce à une fonction d'activation ainsi que les poids et biais associés à chaque neurone. Un neurone fait une somme pondérée des résultats de chaque neurone de la couche précédente et ajoute un biais. Ensuite on applique la fonction sigmoïde au total et on fait passer cette valeur aux neurones de la couche suivante.

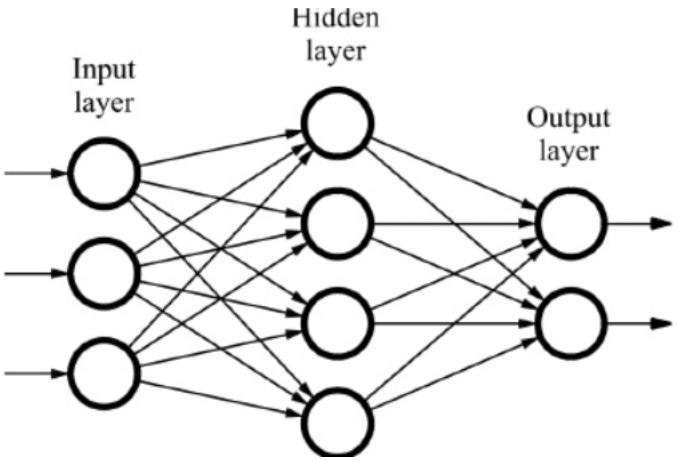

FIGURE 2 – Schéma d'un réseau de neurones à une couche cachée
[3]

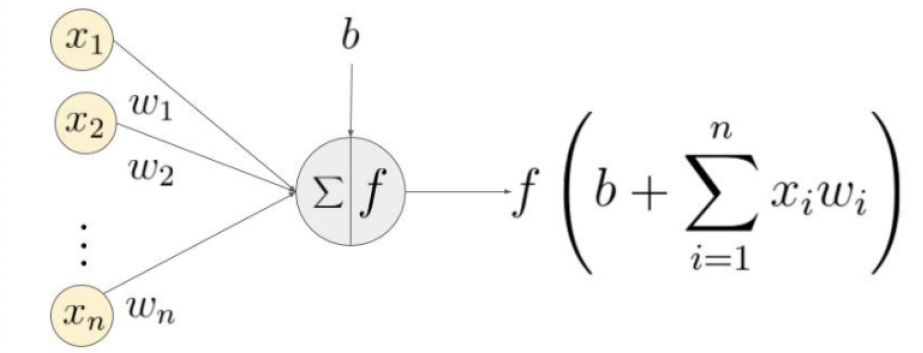

FIGURE 3 – Illustration des poids, biais et fonction d'activation dans un réseau de neurones
[4]

On définit une fonction coût qui mesure la distance entre le résultat de sortie et le résultat attendu. L'objectif est de minimiser notre fonction coût afin que le réseau de neurones puisse approximer le résultat attendu pour un vecteur donné en entrée.
Pour un échantillon de taille N : $(x_{(i)})_{i=1}^{N}$, avec $y(x_{(i)})$ la sortie attendu pour $x_{(i)}$, et $a_L(x_{(i)})$ la sortie à la couche L d'un réseau de neurones à L couches, on définit la fonction coût comme suit :

$$\frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|y(x_{(i)}) - a_L(x_{(i)})\|_2^2 \tag{2}$$

La fonction coût dépend des poids et biais de notre réseau de neurones. Le problème de minimisation consiste donc à trouver les valeurs des poids et biais qui minimisent notre fonction coût. La méthode de minimisation qu'on utilise est le gradient stochastique. L'idée ici est de procéder comme si on appliquait la descente de gradient sauf qu'on ne prend pas en compte tous les points d'entrainement. On calcule le gradient pour un sous ensemble de points d'entrainement choisi de manière stochastique. Le but est de réduire le nombre de calculs surtout pour les cas où on aurait un nombre de points d'entrainement très élevé. L'avantage principal de cette approche est la réduction de temps de calcul à chaque itération. Cependant, la descente n'est pas garantie à chaque coup.

La back propagation nous permet de calculer les dérivées partielles dont on a besoin pour la méthode de gradient stochastique. De manière intuitive, la fonction coût mesure la différence entre la sortie de notre réseau et le résultat attendu. Ceci nous donne une indication sur quels neurones de la couche de sortie il faudrait faire des modifications pour minimiser l'écart avec le résultat attendu. Mais il est important de noter que chaque neurone de la dernière couche dépend des neurones de la couche précédente et ainsi de suite jusqu'à la couche d'entrée. La back propagation est un procédé récursif qui nous permet de modifier les coefficients de notre réseau afin de mieux approximer notre résultat.

Nous allons désormais algorithmiser cette méthode, en s'inspirant par celui proposé par le site *http ://deeplearning.stanford.edu/* [5].
On se donne les notations suivantes :

- N le nombre total de couches, et n_i le nombre de neurone de la i-ème couche (c_i).
- f la fonction d'activation, une sigmoïde.
- x le vecteur des entrées x_i initiales du réseau, de dimension n_1.
- y le vecteur des cibles y_i.
- W la matrice des poids, et b celle des biais, à lire ainsi : pour une couche donnée c, on note $W_{ij}^{(c)}$ ($b_{ij}^{(c)}$) les poids (biais) associés à la connection entre le neurone j de la couche c et celui i de la couche $c+1$.
- a la matrice des sorties de couches, que l'on initialise par $a_i^{(1)} = f(\sum_{j=1}^{n_1} W_{ij}^{(1)} x_j + b_i^{(1)})$. Puis, on définit $a_i^{(c)} = f(z_i^{(c)})$, pour c une couche et i un neurone de celle-ci.
- z la matrice des sommes pondérées des entrées, telle que $z_i^{(c+1)} = \sum_{j=1}^{n_c} W_{ij}^{(c)} a_j + b_i^{(c)}$.
- J la fonction coût, ici celle de l'erreur au sens des moindres de carrés. On la définit comme une fonction de W et b.

Le principe de cette méthode est de calculer le gradient de J, et d'apprendre avec celui-ci. En d'autres termes, après chaque itération (ou *epoch*), on met à jour W et b ainsi, pour α une constante dite *pas* :

$$W_{ij}^{(c)} = W_{ij}^{(c)} - \alpha \frac{\partial}{\partial W_{ij}^{(c)}} J(W, b) \tag{3}$$

$$b_i^{(c)} = b_i^{(c)} - \alpha \frac{\partial}{\partial b_i^{(c)}} J(W, b) \tag{4}$$

Nous allons désormais définir l'algorithme :

1. On se donne k points, choisis *aléatoirement* $(x_{p_1}, \ldots, x_{p_k})$: k est un entier, que l'on nomme taille d'échantillon d'apprentissage ou *batch size*.
2. On réalise une *feedforward pass*, c'est-à-dire que l'on calcule les activations successives sur toutes les couches de notre *batch*.
3. Pour chaque neurone i de la couche de sortie c_N, on pose :
   $\delta_i^{(c_N)} = \frac{\partial}{\partial z_i^{(c_N)}} \frac{1}{2} ||y - a^{(c_N)}||^2 = -(y_i - a_i^{(c_N)}) \cdot f'(z_i^{(c_N)})$
4. Puis, pour chaque neurone i des couches d'indice c entre c_1 et c_{N-1}, on pose :
   $\delta_i^{(c)} = (\sum_{j=1}^{n_{c+1}} W_{ji}^{(c)} \delta_j^{(c+1)}) f'(z_i^{(c)})$
5. On calcule les dérivées partielles souhaitées :
   $\frac{\partial}{\partial W_{ij}^{(c)}} J(W, b) = a_j^{(c)} \delta_i^{(c+1)}$
   $\frac{\partial}{\partial b_i^{(c)}} J(W, b) = \delta_i^{(c+1)}$
6. Enfin, on réalise l'étape de *backpropagation*, à l'aide des équations (3) et (4) :
   $W_{ij}^{(c)} = W_{ij}^{(c)} - \alpha a_j^{(c)} \delta_i^{(c+1)}$
   $b_i^{(c)} = b_i^{(c)} - \alpha \delta_i^{(c+1)}$

Cette algorithme ne représente qu'une seule itération du processus : dans la réalité, on le réalise N_{epochs} fois, ou bien jusqu'à atteindre une erreur sous ϵ_{tol}.

1.2 Second article : Approximation by Superpositions of a Sigmoidal Function[6]

Comme dernière étape préliminaire de notre projet, nous avons eu à lire cet article de G. Cybenko, issu du livre Mathematics of control, Signals, and

Systems, paru chez *Springer-Verlag New York Inc* en 1989.
Ce document s'intéresse à la possibilité d'approximer arbitrairement bien des fonctions de décision continues à l'aide d'un réseau neuronal simple couche muni d'une fonction de sortie sigmoïdale. En ce sens, y est démontrée la densité des combinaisons linéaires finies des fonctions sigmoïdales définies ci-après (1.2) dans l'espace des fonctions continues sur l'hypercube unité.

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j) \tag{5}$$

où $x \in \mathbb{R}^n$, $y_j \in \mathbb{R}^n$ $\forall j \in \{1, \ldots, n\}$, et où les $\alpha_j, \theta \in \mathbb{R}$ $\forall j \in \{1, \ldots, n\}$ sont fixés pour $n \in \mathbb{N}$.
La fonction σ est **sigmoïdale**, que l'on définit ainsi :

$$\sigma(t) \longrightarrow \begin{cases} 1 \text{ si t} \longrightarrow \infty \\ 0 \text{ si t} \longrightarrow -\infty \end{cases}$$

Afin de parvenir à ce résultat, la preuve se déroule en plusieurs étapes. En notant I_n le cube unité $[0,1]^n$, $C(I_n)$ l'espace des fonctions continues sur I_n pour la norme infinie notée $\|f\|$. sera d'abord démontré la densité des sommes finies de n'importe quelle fonction dite **discriminatoire** dans $C(I_n)$.

Définition 1.1. On dit que σ est une fonction **discriminatoire** si pour une mesure $\mu \in M(I_n)$ l'espace des mesures régulières finies boréliennes sur I_n

$$\int_{I_n} \sigma(y^T x + \theta_j)\, d\mu x = 0, \qquad \forall y \in \mathbb{R}^n, \theta \in \mathbb{R}$$

implique que $\mu = 0$.

Le premier théorème démontré par l'auteur est le suivant :

Théorème 1.1. *Soit σ une fonction continue et discriminatoire. Il existe $N \in \mathbb{N}$ tel que la somme de la forme $G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$ est dense dans $C(I_n)$. En notant $S \subset C(I_n)$ l'ensemble des fonctions de la forme $G(x)$, on écrit :*
$\forall f \in C(I_n), \forall \epsilon > 0, \exists\, G \in S$ *tel que* $\|G(x) - f(x)\| < \epsilon$, $\forall x \in I_n$.

L'auteur démontre ce théorème par l'absurde. Il suppose que la fermeture de S n'est pas égale à $C(I_n)$ puis il utilise un corollaire du théorème de Hahn-Banach stipulant qu'il existe une forme linéaire bornée sur $C(I_n)$ nulle sur S mais non nulle sur $C(I_n)$. Il utilise ensuite le théorème de représentation de Riesz ainsi que le fait que σ est discriminatoire et continue dans $C(I_n)$ pour

arriver à une contradiction sur une mesure définie sur I_n supposée non-nulle. Il peut donc conclure de la densité de S dans $C(I_n)$.
L'auteur souhaite ensuite montrer que n'importe fonction sigmoïdale bornée et mesurable est discriminatoire, ce qui entraine que n'importe quelle fonction sigmoïdale continue est discriminatoire. La preuve fait appel au théorème de convergence dominée de Lebesgue, ainsi que la densité des fonctions simples (somme dénombrable des fonctions indicatrices sur un intervalle réel) sur $L^\infty(\mathbb{R})$. Il utilise ensuite la transformée de Fourier d'une mesure $\mu \in M(I_n)$ pour conclure sur le caractère discriminatoire d'une fonction sigmoïdale continue.

Lemme 1.1. *Toute fonction bornée, mesurable, sigmoïdale est discriminatoire. En particulier, toute fonction sigmoïdale continue est discriminatoire.*

Ces deux résultats intermédiaires permettent de prouver le théorème suivant qui est d'intérêt fondamental pour les réseaux neuronaux à une couche cachée, sans contraintes sur le nombre de neurones dans cette couche ainsi que la taille des poids associés à ces neurones :

Théorème 1.2. *Soit σ une fonction sigmoïdale continue, alors les sommes finies de la forme :*

$$G(x) = \sum_{j=1}^{N} \alpha_j \sigma(y_j^T x + \theta_j)$$

sont denses dans $C(I_n)$.

Pour démontrer ce théorème, l'auteur combine les résultats du théorème (1.1) et le lemme (1.1). Enfin, l'auteur démontre les implications de ce résultat dans le contexte des régions de décisions au coeur des applications des réseaux neuronaux :
Soit m la mesure de Lebesgue sur I_n partitionné en k ensembles $P_1, P_2, .., P_k$ disjoints. On définit la fonction de décision de la manière suivante :

$$f(x) = j \; si \; x \in P_j, \forall j \in [\![1, k]\!] \tag{6}$$

Cette fonction peut-être vue comme une fonction de décision pour la classification : si $f(x) = j$ alors $x \in P_j$ et on peut classifier x de cette manière. Peut-on approximer cette fonction de décision avec un réseau neuronal contenant une unique cachée ? Le théorème suivant nous apporte la réponse :

Théorème 1.3. *Soit σ une fonction sigmoïdale. Soit f la fonction de décision d'une partition mesurable quelconque, pour une mesure m, de I_n. Pour tout $\epsilon > 0$, il existe une somme finie G(x) de la forme (5) et un certain $D \subset I_n$ tels que $m(D) \geq 1 - \epsilon$ et que*

$$\|G(x) - f(x)\| \leq \epsilon \; pour \; x \in D$$

Il suffit d'utiliser le théorème de Lusin qui assure l'existence d'une fonction continue h qui est telle que $h(x) = f(x)\,\forall x \in D$, et le théorème (1.2) nous permet de trouver une somme finie $G(x)$ de la forme (5) telle que pour tout $x \in D$, $\|G(x) - f(x)\| = \|G(x) - h(x)\| < \epsilon$.
Il est donc possile d'approximer n'importe quelle fonction de décision, mais l'hypothèse de continuité nous met dans la position de prendre des décisions incorrects sur les points à classer. Il est possible de réduire arbitrairement la mesure totale de l'ensemble des points sur lequel on se trompe en réduisant la valeur de *epsilon* dans le théorème précédent.
Cependant, si ces théorèmes permettent d'assurer l'existence d'un réseau neuronal pour approximer exactement n'importe quelle fonction de décision continue à un ensemble de mesure ϵ prêt, il ne donne aucune indication sur le nombres de neurones nécessaires pour réaliser l'approximation pour une précision donnée. Quelles sont les propriétés de la fonction approximée qui jouent un rôle pour déterminer le nombre de neurones nécessaires ? L'auteur évoque la "malédiction de la dimensionnalité" affectant les domaines de la statistique et d'approximation multidimensionnelles pour émettre l'hypothèse qu'il faudrait un nombre gigantesque de neurones en pratique pour satisfaire la précision évoquée dans le théorème (1.3).

2 Simulations numériques

Dans cette partie, nous avons essayé de construire notre propre réseau neuronal sous Python, afin de mieux comprendre les étapes clés effectuées par un réseau de neurones pour générer des régions de décisions. Nous nous intéressons d'abord à la tâche de classification évoquée dans le premier article de notre étude, puis à l'approximation de fonctions régulières et moins régulières dans un second temps. Pour des raisons de performances liées à nos ordinateurs personnels, nous implémentons en priorité des réseaux à une seule couche cachée dont le nombre de neurones variera.

2.1 Description du modèle

Les réseaux neuronaux de classification et de régression suivent une structure en partie commune : celle de l'étape de la *forward pass*. En reprenant les notations de l'algorithme de Descente du Gradient (1.1) , puis en notant N_x la dimension du vecteur colonne d'entrées x, N_y la dimension du vecteur colonne de sorties y, $W_j^{(1)}$ les matrices de poids de dimensions $1\text{x}N_x$ de la couche cachée, et $W^{(2)}$ celle de dimensions $N_y\text{x}n_1$ de la couche de sortie, on a :

$$\mathbf{RN}(x) = f(W^{(2)}a^{(1)} + b^{(2)}), \text{ où } a_j^{(1)} = f(W_j^{(1)}x + b^{(1)})\ \forall j \in \{1, \dots, n_1\} \tag{7}$$

2.2 Classification

Le code utilisé pour construire le réseau neuronal est en annexe 3.1. Nous nous sommes inspirés des notations du premier article de notre étude pour l'implémenter. Comme nous avons tout construit nous-même, il n'est pas optimisé comme peut l'être un réseau neuronal type "boîte noire" que l'on trouve dans des librairies classiques. Cependant il nous a permis d'étudier l'efficacité d'un réseau neuronal construit à la main, ainsi que des limites que nous allons exposer.
L'étude commence par un réseau neuronal paramétré de la manière suivante : une première couche à deux entrées, une seconde (la couche cachée) à n_1 neurones, et la couche de sortie constituée de deux neurones également. Nous disposons d'un ensemble d'entrainement de *nbech* points de R^2 séparés en deux catégories : $(1,0),(0,1)$. Nous fixons un nombre d'itérations *niter* pour entrainer notre réseau. La fonction d'activation est la fonction sigmoïdale définie en (1). Les coefficients des matrices $W_1 \in \mathbb{R}^{n_1 \times 2}$ et $W_2 \in \mathbb{R}^{2 \times n_1}$ ainsi que des biais $b_1 \in \mathbb{R}^{n_1}$ et $b_2 \in \mathbb{R}^2$ sont choisis aléatoirement avant l'entrainement du réseau. La fonction correspondant à ce réseau neuronal est la suivante :

$$\begin{array}{rccc} F & : & \mathbb{R}^2 & \to & \mathbb{R}^2 \\ & & x & \mapsto & \sigma(W_2\sigma(W_1x+b_1)+b_2) \end{array} \tag{8}$$

Nous utilisons la méthode du gradient stochastique expliquée plus haut afin de minimiser la fonction coût (2) liée à notre réseau. Lors de la phase de test, pour choisir la catégorie à laquelle appartient un échantillon présentée en entrée (x_1,x_2), le réseau neuronal utilise la fonction de décision suivante : Si $F_1(x) > F_2(x)$ alors $(x_1,x_2) \in (1,0)$, sinon $(x_1,x_2) \in (0,1)$.
Nous allons visualiser comment le réseau neuronal établit les régions de décisions dans $\mathbb{R}^2$ en modifiant les ensembles d'entrainement, le nombre de neurones dans la couche cachée n_1 ainsi que le nombre d'itérations *niter*. Pour commencer, $nbech = 10$, $n_1 = 4$ et $niter = 10^5$. On peut alors calculer le nombre de paramètres à évaluer : il y a deux vecteurs de biais de longueur 2 et deux matrices de dimension $2 * n_1 = 8$ ici. Au total, il y a donc 20 paramètres à évaluer pour implémenter ce réseau. L'ensemble d'entraînement pour ce modèle est constitué de 10 points de $[0;1] \times [0;2]$ qui nous permettra de produire une région de décision pour les points de $\mathbb{R}^2$ compris dans le voisinage de cet ensemble.

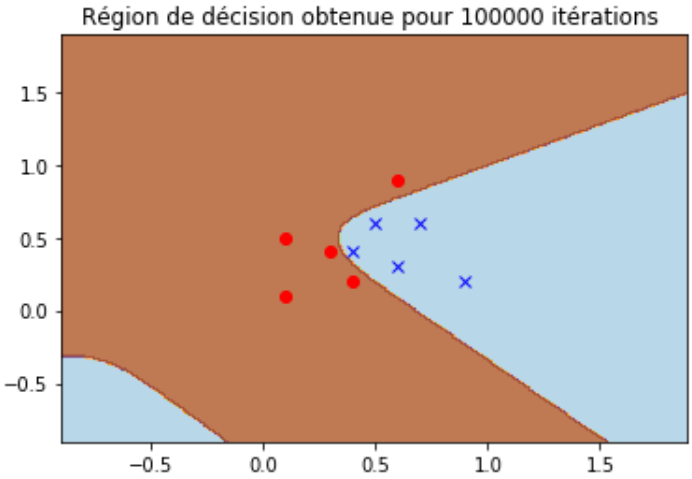

FIGURE 4 – Séparation exacte des échantillons.

La figure 4 nous montre une séparation parfaite des échantillons par notre réseau neuronal. Nous introduisons un point supplémentaire à une position irrégulière pour changer de manière significative la région de décision.

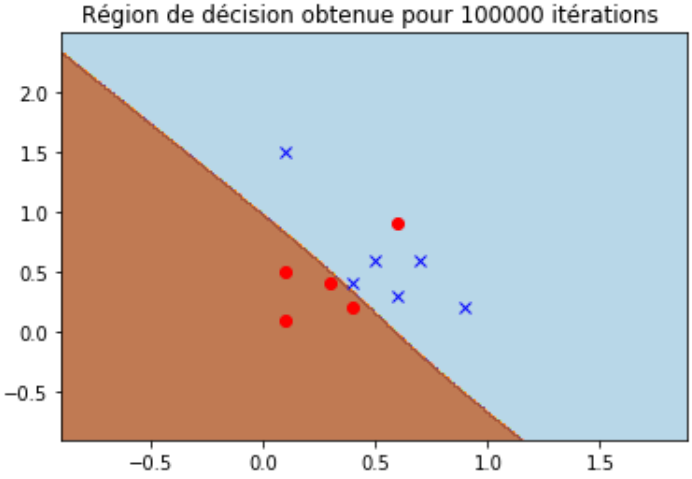

FIGURE 5 – Séparation approximative des échantillons.

Ici on se rend compte que le réseau choisit une séparation linéaire qui ne convient pas à l'ensemble des échantillons (un rond rouge se situe dans la région de décision pour la catégorie des croix bleues). On tente d'augmenter le nombre d'itérations ($niter = 10^6$) pour palier à cette erreur.

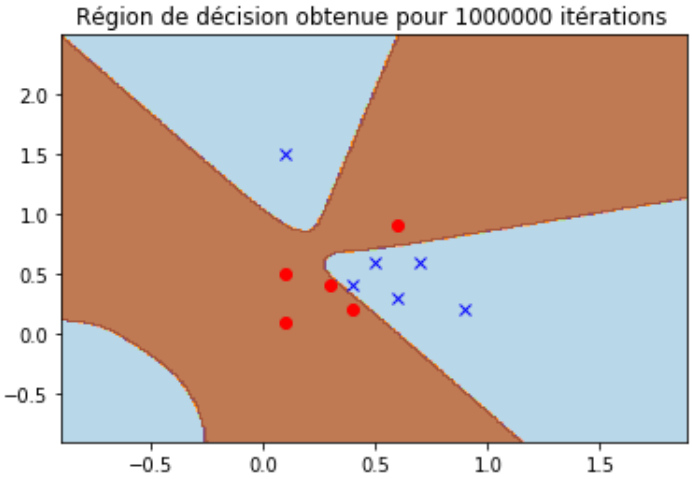

FIGURE 6 – Séparation exacte des échantillons après augmentation du nombre d'itérations.

La méthode fonctionne bien qu'elle soit plus coûteuse en temps de calcul. Nous avons également essayé d'augmenter le nombre de neurones pour résoudre ce problème de classification, mais avec 10^5 itérations, la minimisation de la fonction coût est inefficace et l'on observe aucune convergence vers un minimum. Lorsque l'on rajoute encore un point rouge placé parmi les croix bleues, malgré les $niter = 10^6$ itérations et une convergence vers un plateau minimal de la fonction coût, le réseau ne parvient pas à classer correctement tous les points.

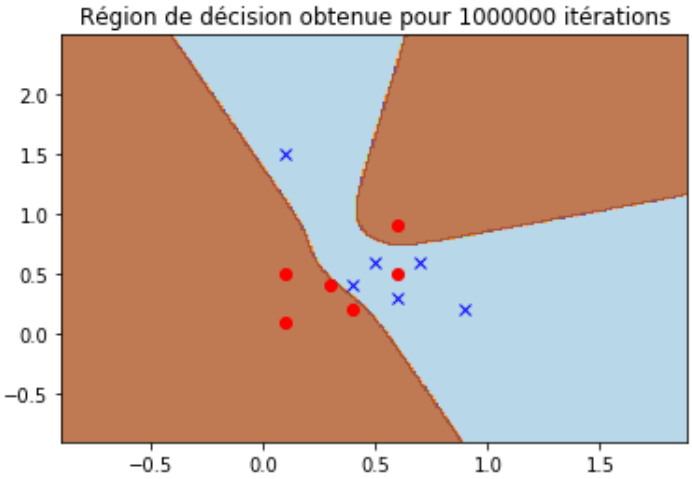

FIGURE 7 – Classification des échantillons avec 10^6 itérations.

On visualise l'évolution de la fonction coût :

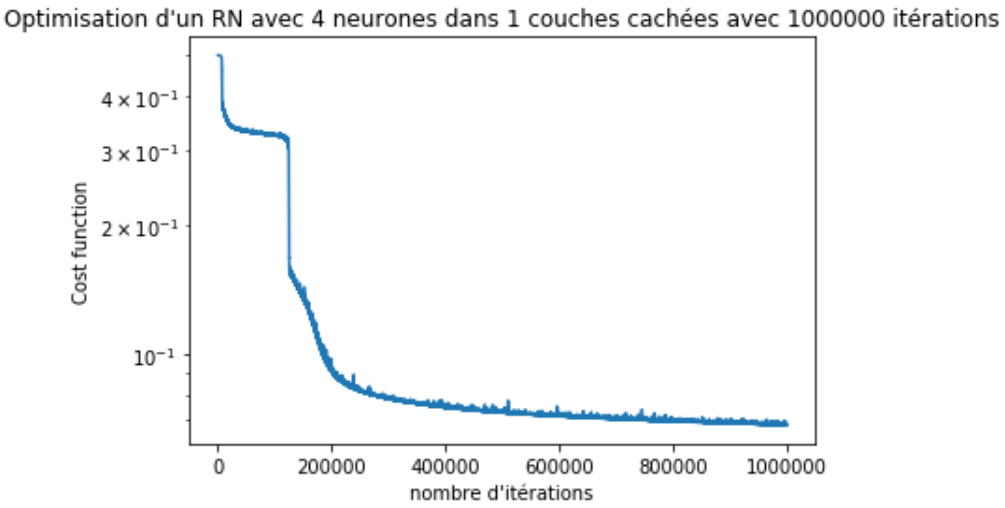

FIGURE 8 – Minimisation effective mais non suffisante de la fonction coût.

La simulation prenant en moyenne 20 minutes sur nos machines, il nous était impossible d'augmenter d'un ordre de grandeur le nombre d'itérations. Nous avons alors décidé d'augmenter le nombres de neurones, mais avec 5 neurones ou plus la précision était moindre :

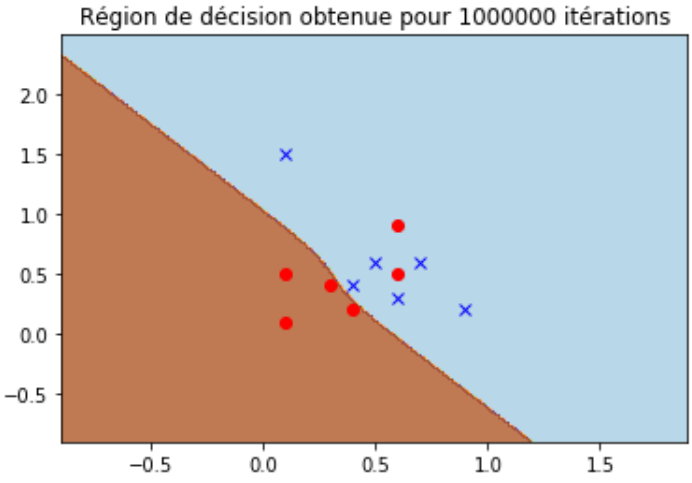

FIGURE 9 – Classification des échantillons avec 5 neurones et 10^6 itérations.

Après avoir effectué une série de tests sur la tâche de classification pour un nombre d'échantillons autour de la dizaine, nous nous sommes fixés $niter = 10^6$ itérations avec $n_1 = 4$ neurones dans la couche cachée comme paramètres optimales pour notre modèle afin de réaliser cette tâche sur nos

machines. Nous avons alors souhaité évaluer notre modèle sur l'approximation de fonctions (contenant un nombre d'échantillons de points à traiter plus important) afin de vérifier en pratique la théorie exposée dans le second article de notre étude avec l'emploi de fonction plus ou moins régulières.

2.3 Régression

2.3.1 Mise en place du Réseau Neuronal

Pour cette partie de notre étude, afin d'avoir un réseau régressif, il nous a fallu modifier notre code de façon à obtenir des coordonées en sortie. Ainsi, nous avons entièrement remanié notre algorithme (en annexe 3.2) afin de suivre au mieux celui fourni dans la partie 1.1, tout en l'exprimant sous la forme d'une **classe** Python, afin d'obtenir une flexibilité maximale lors de l'utilisation. A l'image du précédent réseau, les poids et biais sont initialisés au hasard, avec des valeurs comprises entre 0 et 1.

Cette classe permet à l'utilisateur de décider quant à plusieurs options du réseau, à savoir de sa taille **InfosCC** (sous la forme d'un tuple, tel que $(4, 2)$ par exemple), du nombre maximal d'itérations **Niter** (par défaut 10000), de la graine stochastique **seed** (par défaut 1) ainsi que du pas **pas** (par défaut 0.3). Le réseau est composé d'un nombre de couches égale à la longueur du tuple **InfosCC**, chacune munie d'un nombre de neurones égal aux informations comprises dans ce même tuple. De plus, une ultime couche, dite de sortie, est ajoutée à la fin du réseau, ne comprenant qu'un unique neurone. Nous avons décidé de limiter le nombre de sorties de chaque neurone à une seule, à l'exception de celui de la dernière couche qui contient un nombre de sorties égal à la dimension de la cible renseignée.

La première fonction de cette classe est celle dite de **fit**. Elle demande en entrée l'échantillon d'apprentissage ainsi que sa cible liée, puis entraîne le réseau à l'aide de la méthode du Gradient Stochastique pour un nombre d'itération Niter. La structure de cette fonction est analogue à celle de fit d'un réseau neuronal de classification : simplement, le cibles entrées peuvent être continues.
La seconde fonction est celle dite de **prédiction**. A la différence de son équivalent pour un réseau de classification, il n'y a aucune décision réalisée quant à une classe : le réseau ne retourne simplement que le **score** produit par une itération entraînée du réseau. Cette prise de position amène à un problème d'espace de sortie : en effet, étant donné que la dernière couche demeure une sigmoïde, et non pas une $reLU$ ou autre fonction prenant des valeurs hors de l'intervalle $[0, 1]$, nous ne pouvons pas approximer de fonctions en sortant sans devoir effectuer un travail sur celles-ci.

Enfin, nous avons réalisés quelques essais préliminaires afin d'évaluer notre réseau neuronal regressif. Ces tests ont été réalisés à l'aide d'une initialisation **MCP_Regressor((20,), Niter = 10000, seed = 1, pas = 0.3)**, c'est-à-dire à l'aide d'un réseau à une unique couche de 20 neurones.

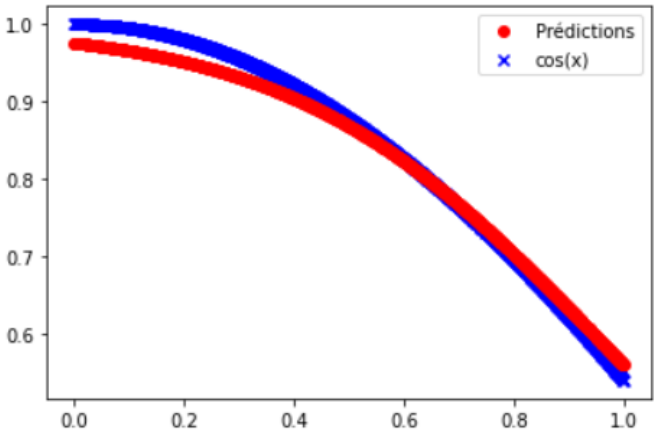

FIGURE 10 – Approximation d'un cosinus sur [0,1]

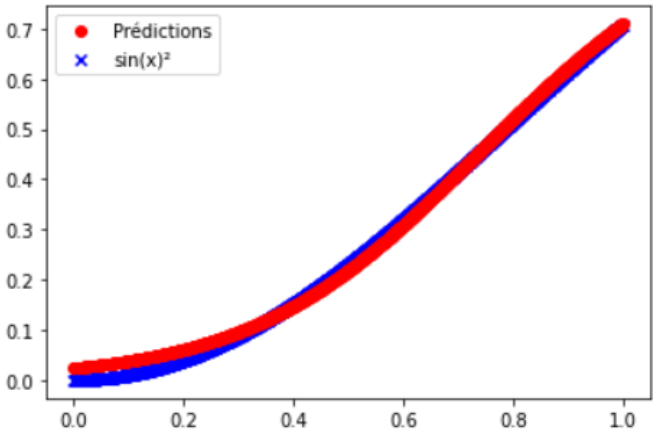

FIGURE 11 – Approximation d'un sinus carré sur [0,1]

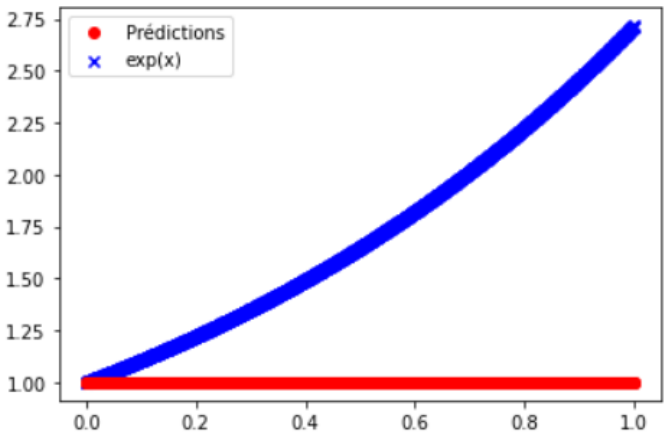

FIGURE 12 – Approximation d'une exponentielle sur [0,1]

Ainsi, on remarque que le réseau approxime relativement correctement des fonctions continues et monotones à valeurs dans $[0, 1]$. En revanche, il rend des résultats absurdes quand on tente de sortir de cet intervalle : en effet, il ne peut approximer correctement l'exponentielle, car une sigmoïde ne peut pas dépasser 1.

2.3.2 Etude de l'expressivité

Nous avons choisi de faire des régressions sur deux fonctions :

$$f(x_i) = \sin(x_i) + \epsilon_i \tag{9}$$

où $(\epsilon_i)_i$ est un bruit gaussien

$$g(x_i) = a\Pi(x_i - 0.5) + b \tag{10}$$

où Π est la fonction porte (créneau) qu'on translate de 0.5, a et b des coefficients qui permettent d'avoir une amplitude dans]0,1[

Pour la fonction sinus bruitée, l'idée est de simuler des données réelles. Ces dernières sont rarement propres et l'objectif est de récupérer notre sinus de base grâce à notre régression. Pour la fonction porte, ce qui nous intéresse le plus est la discontinuité de notre fonction. On veut voir comment se comporte notre réseau pour approximer les zones de discontinuité étant donné que dans l'article [6] on ne traite que l'approximation de fonctions continues dans l'hypercube [0,1]. Il est donc intéressant de voir les limites de notre approximation lorsqu'on a des zones de discontinuité. Nous avons étudié l'expressivité de notre réseau de neurones en modifiant le nombre de neurones, le nombre de couches cachées ainsi que le nombre d'itérations.

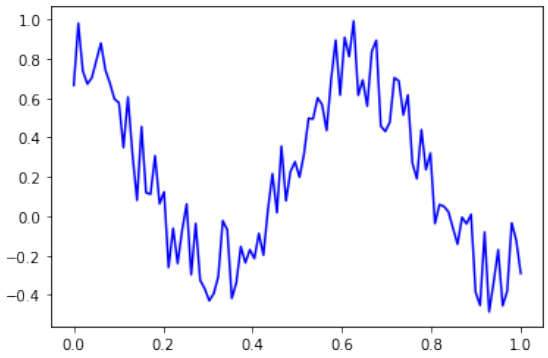

FIGURE 13 – Sinus bruité

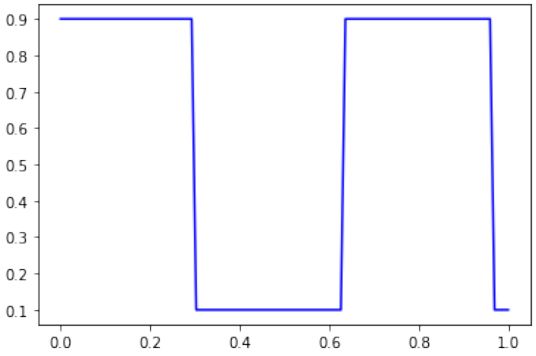

FIGURE 14 – Fonction créneau

Les titres des illustrations ci-dessous indiquent l'architecture des couches internes ainsi que le temps de calcul.

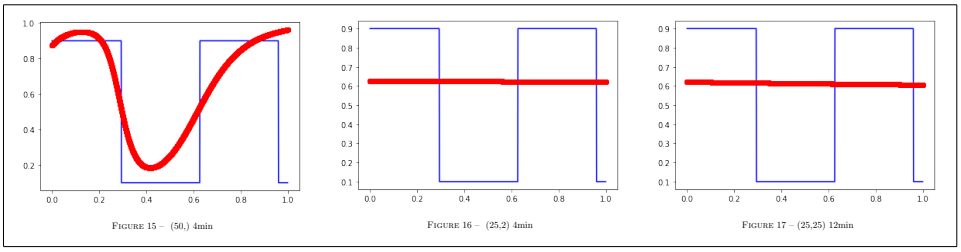

FIGURE 15 – (50,) 4min

FIGURE 16 – (25,2) 4min

FIGURE 17 – (25,25) 12min

TABLE 1 – Simulations à 10^5 itérations

Pour la fonction créneau, l'architecture $(50,)$ semble être la plus performante pour $niter = 10^5$.

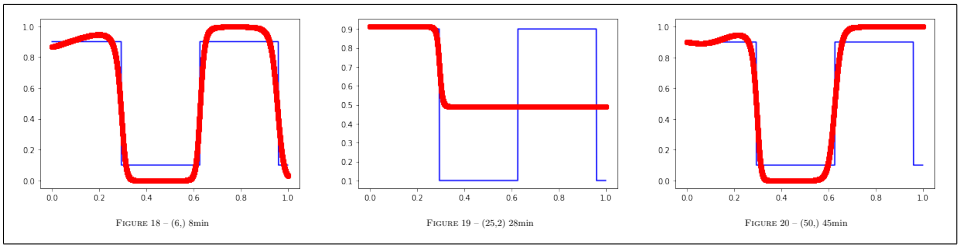

FIGURE 18 – (6,) 8min FIGURE 19 – (25,2) 28min FIGURE 20 – (50,) 45min

TABLE 2 – Simulations à 10^6 itérations

Pour $niter = 10^6$, nous observons un résultat satisafaisant avec une architecture $(6,)$ alors que les simulations avec plus de neurones donnent pas de meilleurs résultats.

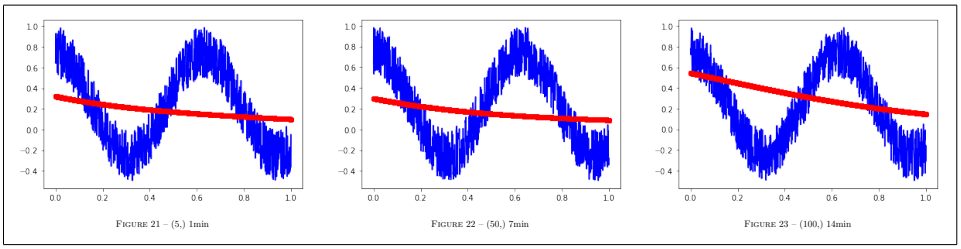

FIGURE 21 – (5,) 1min FIGURE 22 – (50,) 7min FIGURE 23 – (100,) 14min

TABLE 3 – Simulations à 10^5 itérations

Ici, nous observons que pour toutes les configurations, on obtient une allure similaire. Le nombre d'itérations semble être insuffisant.

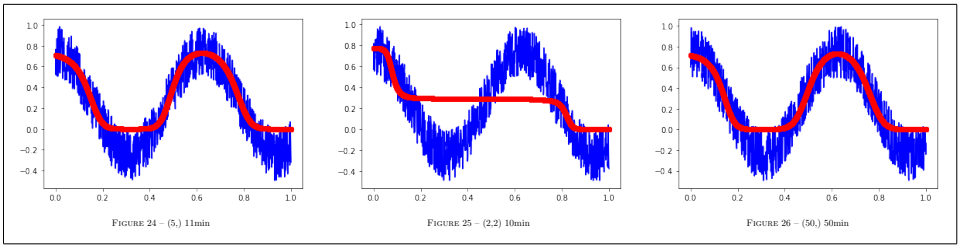

FIGURE 24 – (5,) 11min FIGURE 25 – (2,2) 10min FIGURE 26 – (50,) 50min

TABLE 4 – Simulations à 10^6 itérations

Pour $niter = 10^6$, nous observons de bons résultats pour un nombre de neurones modeste. Deux couches cachées à peu de neurones ne semblent pas efficaces.

Les simulations qu'on a réalisées nous indiquent que généralement, le facteur le plus important dans une régression par réseaux de neurones est le nombre d'itérations. Pour avoir une allure correcte il faut un nombre d'itérations de l'ordre de 10^6. L'architecture du réseau de neurones et le nombre de couche cachées est moins évident à étudier. On peut faire la remarque que pour la régression, une couche cachée marche assez bien. L'ajout de couches cachées supplémentaires ne garantit pas un meilleur résultat, mais rajoute du temps de calcul. On s'apperçoit également que même dans le cadre de la fonction créneau, on obtient une régression continue. Ceci s'explique par le fait qu'on a une fonction d'activation sigmoïdale qui d'après l'article de Cybenko [6] approxime des fonctions continues.

3 Annexes

3.1 Code du Réseau Neuronal de Classification

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Apr 21 15:53:40 2021

@author: theophilebaggio
"""


import numpy as np
from math import *
import random as rd
import matplotlib.pyplot as plt

def activate(x,W,b):
    M = 1/(1 + np.exp(-(W.dot(x) + b)))
    return M

def cost_function(Weight_mat_list,bias_list,X,Y):
    costvec = np.zeros(len(X[0]))
    for i in range(0,len(X[0])):
        x = X[:,i].reshape(len(X),1)
        y = Y[:,i].reshape(len(Y),1)
        a = x
        for j in range(len(bias_list)):
            a = activate(a,Weight_mat_list[j],bias_list[j])
```

```
        costvec[i] = np.linalg.norm(y-a,2)
    costvalue = 1/len(costvec) * 1/2 * np.linalg.norm(costvec,2)**2
    return costvalue

def predict(Weight_mat_list,bias_list,X):
    prediction_list = []
    for i in range(len(X[0])):
        a = X[:,i].reshape(len(X),1)
        for j in range(len(bias_list)):
            a = activate(a,Weight_mat_list[j],bias_list[j])
        if(a[0]<=a[1]):
            prediction_list.append([0,1])
        else:
            prediction_list.append([1,0])
    return np.array(prediction_list).T


## MAIN PROGRAM
#data
donnees = np.array([[0.1,0.3,0.1,0.6,0.4,0.6,0.5,0.9,0.4,0.7,0.1,0.6],\
                    [0.1,0.4,0.5,0.9,0.2,0.3,0.6,0.2,0.4,0.6,1.5,0.5]])
target = np.array([[1,1,1,1,1,0,0,0,0,0,0,1],[0,0,0,0,0,1,1,1,1,1,1,0]])

nb_layers = 1
liste_nb_neurons = []
liste_weigth_mat = []
liste_bias = []

for i in range(nb_layers):
    n = i+1
    neurons = int(input('How many neurons would you like\
 in the hidden layer n  {} ?'.format(n)))
    liste_nb_neurons.append(neurons)

nb_n_hid_lay = sum(liste_nb_neurons)
nbinput = 2
nboutput = 2
liste_nb_neurons.append(nboutput)
liste_nb_neurons.insert(0,nbinput)

for nb in range(nb_layers+1):
    n_current_layer = liste_nb_neurons[nb+1]
    n_previous_layer = liste_nb_neurons[nb]#nb de neurones dans la hidden
                                                                #layer
```

```
    np.random.seed(5);
    weight_mat  = 1/2 * np.random.randint(10,\
                                          size=(n_current_layer,
                                                n_previous_layer))
    bias=1/2 * np.random.randint(10, size=(n_current_layer,1))
    liste_weigth_mat.append(weight_mat)
    liste_bias.append(bias)

##initialisation des param tres du gradient stochastique
eta = 0.05 #learnig rate
Niter = 1000000 #nombre d'iterations maximum
savecost = np.zeros((Niter))

for i in range(0,Niter):

    #feedforward
    k = np.random.randint(len(donnees[0]))
    x = donnees[:,k].reshape(nbinput,1)
    activation_vect = [x]
    d_activation_vect = []
    mat_da_vect = []

    for j in range(nb_layers + 1):
        a = activate(activation_vect[j],liste_weigth_mat[j],\
                                            liste_bias[j])
        activation_vect.append(a)
        d_activation_vect.append(a*(1-a))
        mat_da_vect.append(np.diag(d_activation_vect[j][:,0]))

    #backpropagation
    liste_delta = []
    delta_out = mat_da_vect[-1].dot\
        ((activation_vect[-1]-target[:,k].reshape(nboutput,1)))
    liste_delta.append(delta_out)

    for j in range(nb_layers):
        delta = mat_da_vect[-1 -(j+1)].dot\
            (np.transpose(liste_weigth_mat[-1 - j]).dot(liste_delta[j]))
        liste_delta.append(delta)
    liste_delta.reverse()

    #on ajuste les coefficients
    for j in range(len(liste_delta)):
        a = activation_vect[-1-(j+1)]
```

```
            prov_value = liste_delta[-1-j].dot(np.transpose(a))
            mat = liste_weigth_mat[-1-j]
            liste_weigth_mat[-1-j] = mat - eta*prov_value
            liste_bias[-1-j] = liste_bias[-1-j]- eta*liste_delta[-1-j]

        #on sauve la valeur de la cost function apr s chaque iteration de
        #la methode du gradient stochastique
        savecost[i] = cost_function(liste_weigth_mat,\
                                    liste_bias,  donnees, target)

predictions = predict(liste_weigth_mat,liste_bias,donnees)
print(predictions)

plt.figure(2)
plt.semilogy(savecost)
plt.xlabel("nombre d'it rations")
plt.ylabel('Cost function')
plt.title("Optimisation d'un RN avec {} neurones dans\
 {} couche cach e".format(nb_n_hid_lay,nb_layers))


#on essaie de dessiner la s paration par zone apprise par le reseau
                                                    #neuronal
x1 = donnees[0]
x2 = donnees[1]
y = target
h = 0.01
x_min, x_max = x1.min() - 1, x1.max() + 1
y_min, y_max = x2.min() - 1, x2.max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))

Z = predict(liste_weigth_mat,liste_bias,np.c_[xx.ravel(), yy.ravel()].T)

plt.figure(3)
for i in range(len(x1)):
        if(y[1,i]==1):
            plt.plot(x1[i],x2[i],'bx')
        else:
            plt.plot(x1[i],x2[i],'ro')

Z = Z[0].reshape(xx.shape)
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)
plt.title("R gion de d cision obtenue pour
```

```
                    {} iterations".format(Niter))
plt.show()
```

3.2 Code du Réseau Neuronal de Régression

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import norm
from tqdm.notebook import tqdm # Permet d'afficher
    #une barre de chargement.



class MCP_Regresseur :

    def __init__(self, InfosCC, Niter = 10000, seed = 1, pas = 0.3):
        """
        Cette fonction va permettre d'assigner dans des variables
        locales    la classe les informations sur le MCP
        """
        # On initialise les couches
        self.N_CC = len(InfosCC) + 1
        self.T_CC = np.ones(self.N_CC, dtype = int)
        for couche in range(self.N_CC - 1) :
            self.T_CC[couche] = InfosCC[couche] # Dans la derniere
            couche, on a un seul neurone

        # On recupere les autres informations
        self.Niter = Niter
        self.seed = seed
        self.pas = pas


    def activation(self, x, Coef, Biais) :
        """
        Fonction d'activation de la fonction. Ici une sigmoide
        """
        A = 1 / (1 + np.exp( - (Coef.dot(x) + Biais)))
        return A


    def fit(self, data, target):
        """
        Fonction d'apprentissage sur un couple (data,target)
```

```
"""

# Pour une ecriture plus simple, on recupere les variables
locales a la classe qui seront les plus appelees
N_CC = self.N_CC
T_CC = self.T_CC
pas = self.pas

# On s'interesse aux donnees fournies
Dim_E = np.ones(N_CC, dtype = int)
if (data.ndim == 1):
    Dim_E[0] = 1
else :
    Dim_E[0] = data.shape[1]
Dim_E[1:] = T_CC[0:-1]

Dim_S = np.ones(N_CC, dtype = int)
if (target.ndim == 1):
    Dim_S[-1] = 1
else :
    Dim_S[-1] = target.shape[1]

T_ech = len(data)

# On genere les matrices de Coefficients de Poids
Coefs = {} # On va utiliser un dictionnaire pour une lecture
           #plus aisee

for couche in range(N_CC):
    for neurone in range(T_CC[couche]):
        Coefs[couche,neurone] = 0.5 *
        np.random.rand(Dim_S[couche], Dim_E[couche])

# On genere les vecteurs de Biais
Biais = {}

for couche in range(N_CC):
    for neurone in range(T_CC[couche]):
        Biais[couche,neurone] = 0.5 *
                        np.random.rand(Dim_S[couche], 1)

# Etapes de Convergence
barre = tqdm(total = self.Niter) # Initialisation de la barre
        #de progression
```

```
for epoch in range(self.Niter):
    barre.update(1) # La barre avance

    k = np.random.randint(T_ech) # Methode stochastique
    x = data[k].reshape(-1,1)
    y = target[k].reshape(-1,1)

    # Propagation avant
    a = {} # On stocke les sorties de neurones

    x_temp = x
    for couche in range(N_CC):
        if (couche != (N_CC - 1)):
            x_newtemp = np.zeros((T_CC[couche],1))
            for neurone in range(T_CC[couche]):
                a[couche,neurone] = self.activation(x_temp,
                Coefs[couche,neurone], Biais[couche,neurone])
                x_newtemp[neurone] = a[couche,neurone]
            x_temp = x_newtemp.reshape(-1,1)
        else :
            for neurone in range(T_CC[-1]):
                a[couche,neurone] = self.activation(x_temp,
                Coefs[couche,neurone], Biais[couche,neurone])


    # Calcul du gradient
    delta = {}

    for couche in range(N_CC):
        couche_retro = N_CC - couche - 1

        for neurone in range(T_CC[couche_retro]):
            a_temp = a[couche_retro,neurone]
            derivee = (a_temp * (1 - a_temp))

            if (couche == 0):
                delta[couche_retro,neurone] = -(y -
                a_temp).T.dot(derivee)

            elif (couche == 1):
                for neurone_retro in range(T_CC[couche_retro +
                1]):
                    sum_d = Coefs[couche_retro +
```

```
                    1,neurone_retro][0,neurone] *
                    delta[couche_retro + 1,neurone_retro]
                sum_d = np.sum(sum_d)

                delta[couche_retro,neurone] = sum_d * derivee

            else :
                for neurone_retro in range(T_CC[couche_retro +
                1]):
                    sum_d = Coefs[couche_retro + 1,neurone_retro]
                    delta[couche_retro + 1,neurone_retro]
                sum_d = np.sum(sum_d)

                delta[couche_retro,neurone] = sum_d * derivee


    # Descente

    x_temp = x
    for couche in range(N_CC):
        if (couche != (N_CC - 1)):
            x_newtemp = np.zeros((T_CC[couche],1))
            for neurone in range(T_CC[couche]):
                Coefs[couche,neurone] = Coefs[couche,neurone] -
                pas * delta[couche,neurone].dot(x_temp.T)
                Biais[couche,neurone] = Biais[couche,neurone] -
                pas * delta[couche,neurone]
                x_newtemp[neurone] = a[couche,neurone]
            x_temp = x_newtemp

        else :
            for neurone in range(T_CC[couche]):
                Coefs[couche,neurone] = Coefs[couche,neurone] -
                pas * delta[couche,neurone].dot(x_temp.T)
                Biais[couche,neurone] = Biais[couche,neurone] -
                pas * delta[couche,neurone]

    self.Coefs_ = Coefs
    self.Biais_ = Biais

    # On recupere le nombre de labels de la target

    if (target.ndim == 1):
        Labels = np.unique(target)
```

```
            else :
                Labels = np.unique(target.reshape(-1,1), axis = 1)
            self.Labels_ = Labels

            self.T_Labels = Dim_S[-1]


    def predict(self, data):
        """
        Fonction de prediction.
        """

        # On recupere les variables qui seront frequemment utilisees
        T_CC = self.T_CC
        N_CC = self.N_CC
        Coefs = self.Coefs_
        Biais = self.Biais_

        T_ech = len(data)

        # Propagation avant
        a = {} # On stocke les sorties de neurones
        predictions = np.zeros((T_ech))

        for indice in range(T_ech):
            x_temp = data[indice].reshape(-1,1)
            for couche in range(N_CC):
                if (couche != (N_CC - 1)):
                    x_newtemp = np.zeros((T_CC[couche],1))
                    for neurone in range(T_CC[couche]):
                        a[couche,neurone] = self.activation(x_temp,
                        Coefs[couche,neurone], Biais[couche,neurone])
                        x_newtemp[neurone] = a[couche,neurone]
                    x_temp = x_newtemp.reshape(-1,1)
                else :
                    for neurone in range(T_CC[-1]):
                        a[couche,neurone] = self.activation(x_temp,
                        Coefs[couche,neurone], Biais[couche,neurone])
            predictions[indice] = a[N_CC - 1,0]
        return(predictions)
```

Bibliographie

[1] Catherine F. Higham et Desmond J. Higham. "Deep Learning : An Introduction for Applied Mathematicians". In : (2018). arXiv : 1801.05894 [math.HO].

[2] Sigmoid function. URL : https://en.wikipedia.org/wiki/Sigmoid_function.

[3] Feed-Forward Neural Network : URL : https://databricks.com/fr/glossary/neural-network.

[4] Understanding Activation Functions in Deep Learning. URL : https://learnopencv.com/understanding-activation-functions-in-deep-learning/.

[5] Unsupervised Feature Learning and Deep Learning Tutorial. URL : http://deeplearning.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/.

[6] G. Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In : Mathematics of Control, Signals, and Systems (1989), p. 303-314.